Die Boost C++ Bibliotheken

Boris Schäling

25. April 2015

Die Boost C++ Bibliotheken
by Boris Schäling

Edition Zweite deutsche
Published 25. April 2015

Inhaltsverzeichnis

Vorwort

Was Sie lernen werden

Dieses Buch führt Sie in zahlreiche Boost-Bibliotheken ein, die die Standardbibliothek um in der Praxis nützliche Funktionen ergänzen. Da die Boost-Bibliotheken auf dem Standard basieren, sind sie in modernstem C++ entwickelt. Die Boost-Bibliotheken sind plattformunabhängig und werden auf vielen Betriebssystemen inklusive Windows und Linux von einer großen Entwicklergemeinschaft unterstützt.
Die Boost-Bibliotheken ermöglichen es Ihnen, Ihre Produktivität als C++-Entwickler zu steigern. So können Sie zum Beispiel auf Smartpointer zugreifen, mit denen Sie ohne viel Mühe zuverlässigeren Code schreiben können, oder eine Bibliothek verwenden, mit der Sie plattformunabhängige Netzwerkanwendungen entwickeln können. Da die Boost-Bibliotheken Entwicklungen im Standard teilweise vorwegnehmen, können Sie früher von Hilfsmitteln profitieren, ohne auf deren abschließende Standardisierung und Verfügbarkeit in Implementationen der Standardbibliothek zu warten.

Was Sie wissen müssen

Die Boost-Bibliotheken basieren auf dem Standard. Sie sollten daher den Standard ausreichend gut kennen. So sollten Sie Container, Iteratoren und Algorithmen einsetzen können und idealerweise von Konzepten wie RAII, Funktionsobjekten und Prädikaten gehört haben. Je besser Sie den Standard in all seinen Facetten kennen, umso mehr werden Sie von den Boost-Bibliotheken profitieren können.
Sie benötigen grundsätzlich keine Kenntnisse in der Template-Metaprogrammierung, um die Bibliotheken, die in diesem Buch vorgestellt werden, verwenden zu können. Der Schwerpunkt dieses Buchs liegt auf Bibliotheken, die schnell und einfach erlernt werden können und Ihnen in Ihrer Arbeit als C++-Entwickler sofort von großem Nutzen sind.
Viele Beispiele verwenden Features, die mit C++11 in den Standard aufgenommen wurden. So wird zum Beispiel das Schlüsselwort `auto` verwendet, um Typen nicht explizit angeben zu müssen. Konstruktoren werden über die uniforme Initialisierung aufgerufen: Variablen werden wenn möglich mit einem Paar geschweifter statt runder Klammern initialisiert. Viele Beispiele nutzen Lambda-Funktionen, um den Code kürzer und kompakter zu machen. Während Sie die Beispiele auch ohne Detailwissen zu C++11 verstehen können, basiert dieses Buch grundsätzlich auf C++11.

Typographische Konventionen

In diesem Buch werden folgende Formatierungen verwendet:

`Nichtproportionale Schrift` Nichtproportionale Schrift wird für Klassennamen, Funktionsnamen und Schlüsselwörter verwendet – für alles, was nach C++-Code aussieht. Sie wird auch für Code-Beispiele, Kommandozeilenparameter und die Datenausgabe von Programmen verwendet. Zum Beispiel: `int i = 0;`

`Nichtproportionale fette Schrift` Nichtproportionale fette Schrift wird für Variablennamen, Objekte und Benutzereingaben verwendet. Zum Beispiel: Die Variable **`i`** wird mit der Zahl 0 initialisiert.

Fett Befehle werden fett markiert. Zum Beispiel: Die Boost-Bibliotheken werden mit einem Programm namens **bjam** kompiliert.

Kursiv Kursive Schrift wird verwendet, wenn neue Konzepte eingeführt und zum ersten Mal erwähnt werden. Zum Beispiel: *RAII* ist die Abkürzung von resource acquisition is initialization – einem Konzept, auf dem Smartpointer basieren.

Beispiele

Dieses Buch enthält mehr als 430 Beispiele. Jedes Beispiel ist vollständig und kann von Ihnen kompiliert und ausgeführt werden. Sie können alle Beispiele von http://dieboostcppbibliotheken.de/beispiele herunterladen, um schneller loslegen zu können.
Alle Beispiele wurden mit folgenden Compilern getestet:

- Microsoft Visual C++ 2013 Update 1 (64-Bit Windows 7 Professional mit Service Pack 1)
- GCC 4.9.2 (64-Bit Cygwin 1.7.34)
- GCC 4.6.3 (32-Bit Ubuntu 12.04.4)
- Clang 3.3 (32-Bit Ubuntu 12.04.4)

Alle Beispiele in diesem Buch basieren auf C++11. Die Compiler-Unterstützung für C++11 war, soweit notwendig, aktiviert.
Einige Beispiele in diesem Buch sind plattformspezifisch. In den Beschreibungen der Beispiele wird darauf hingewiesen, wenn diese lediglich unter Windows oder einem anderen Betriebssystem funktionieren.
Alle Beispiele werden so, wie sie sind, bereitgestellt, ohne ausdrückliche oder implizite Garantie. Sie sind unter der Boost Software Lizenz veröffentlicht.

Einführung

Die Boost C++ Bibliotheken sind eine Sammlung moderner, auf dem C++-Standard basierender Bibliotheken. Ihr Quellcode steht unter der Boost Software Lizenz zur Verfügung, die es gestattet, die Bibliotheken kostenlos einzusetzen, zu verändern und zu verbreiten. Die Bibliotheken sind plattformunabhängig und unterstützen neben weitverbreiteten Compilern auch zahlreiche weniger bekannte Compiler.
Verantwortlich für die Entwicklung und Veröffentlichung der Boost-Bibliotheken ist die Boost-Community. Es handelt sich hierbei um eine relativ große Gruppe an Entwicklern aus aller Welt, die sich online über die Website www.boost.org und Mailinglisten koordinieren. Als Code-Repository wird GitHub verwendet. Ziel dieser Community ist es, qualitativ hochwertige Bibliotheken zu entwickeln und zu sammeln, die die Standardbibliothek ergänzen. Bibliotheken, die sich in der Praxis bewähren und in der Entwicklung von C++-Programmen eine große Bedeutung erlangen, besitzen gute Chancen, eines Tages in die Standardbibliothek aufgenommen zu werden.
Die Boost-Community entstand um das Jahr 1998 herum, als die erste Version des Standards veröffentlicht wurde. Sie ist seitdem kontinuierlich gewachsen und spielt eine große Rolle in der Standardisierung von C++. Auch wenn es keine direkte Beziehung zwischen der Boost-Community und dem Standardisierungsgremium von C++ gibt, sind einige Entwickler in beiden Gruppen aktiv. So wurden mit C++11 einige Bibliotheken in den Standard aufgenommen, die ihren Ursprung in der Boost-Community hatten.
Die Boost-Bibliotheken bieten sich vor allem dann an, wenn Sie den Standard ausgeschöpft haben und nach neuen Möglichkeiten suchen, die Produktivität in Ihren Projekten zu steigern. Da die Boost-Bibliotheken schneller weiterentwickelt werden als der Standard, haben Sie früher Zugriff auf Neuentwicklungen und müssen nicht darauf warten, bis diese in eine neue Version der Standardbibliothek aufgenommen werden. So können Sie schneller von aktuellen Entwicklungen in C++ profitieren und diese in Ihren Projekten verwenden.
Aufgrund des hervorragenden Rufs, den die Boost-Bibliotheken genießen, stellen Kenntnisse im Umgang mit den Bibliotheken einen beruflichen Vorteil dar. So ist es zum Beispiel nicht ungewöhnlich, in Bewerbungsgesprächen nach Kenntnissen zu den Boost-Bibliotheken gefragt zu werden. Von Entwicklern, die die Boost-Bibliotheken kennen, kann eher erwartet werden, dass sie über aktuelle Entwicklungen in C++ Bescheid wissen und modernen C++-Code schreiben und verstehen können.

Entwicklungsprozess

Die Entwicklung von Boost-Bibliotheken ist nur möglich, wenn Entwickler oder Organisationen diese mit großem Engagement betreiben. Weil nur Bibliotheken von Boost akzeptiert werden, die in der Praxis existierende Probleme lösen, ein überzeugendes Design besitzen, in modernem C++ entwickelt sind und darüber hinaus verständlich dokumentiert sind, steckt in jeder einzelnen Boost-Bibliothek viel Arbeit.
Grundsätzlich steht es jedem Entwickler frei, sich in der Boost-Community zu engagieren und neue Bibliotheken vorzuschlagen. Damit aus einer Idee eines Tages eine Boost-Bibliothek wird, ist es jedoch unausweichlich, selbst viel Zeit und Arbeit zu investieren. Dabei ist es von Vorteil, in den Boost-Mailinglisten mit anderen Entwicklern und potentiellen Anwendern über Anforderungen und Lösungen zu diskutieren.
Neben Boost-Bibliotheken, die tatsächlich aus dem Nichts erschaffen werden, können auch existierende Bibliotheken zur Integration in Boost vorgeschlagen werden. Da für diese Bibliotheken die gleichen Anforderungen gelten wie für Bibliotheken, die speziell zur Integration in Boost entwickelt wurden, sind jedoch unter Umständen größere Änderungen erforderlich.
Ob eine Bibliothek in Boost aufgenommen wird oder nicht, hängt vom Ausgang eines Reviews ab. Entwickler können für ihre Bibliotheken ein Review beantragen, das rund zehn Tage dauert. Während dieses Zeitraums sind andere Entwickler aufgefordert, eine Bibliothek zu bewerten. Je nachdem, wie viele positive und negative Bewertungen eingehen, entscheidet der Review Manager, ob die Bibliothek in Boost aufgenommen wird oder nicht.
Da einige Entwickler während eines Reviews unter Umständen zum ersten Mal mit einer Bibliothek arbeiten, ist es nicht ungewöhnlich, wenn es während des Reviews Änderungswünsche an der Bibliothek gibt.

Sollte eine Bibliothek aus technischen Gründen abgelehnt werden, ist es durchaus möglich, die Bibliothek zu überarbeiten und für eine neue Version ein neues Review zu beantragen. Wird eine Bibliothek abgelehnt, weil sie keine praxisrelevanten Probleme löst oder eine nicht überzeugende Lösung für ein praxisrelevantes Problem bietet, wird sie wahrscheinlich auch in einem neuen Review durchfallen.
Nachdem jederzeit neue Bibliotheken in Boost aufgenommen werden können, wird alle drei Monate eine neue Version der Boost-Bibliotheken veröffentlicht. Dies stellt sicher, dass Entwickler von Verbesserungen in den Boost-Bibliotheken regelmäßig und zeitnah profitieren.

> Anmerkung
>
> Zum Abschluss dieses Buchs war 1.57.0 die aktuelle Version der Boost-Bibliotheken. Diese Version wurde im November 2014 veröffentlicht.

Installation

Die Boost-Bibliotheken stehen als Quellcode zur Verfügung. Während die meisten dieser Bibliotheken ausschließlich aus Headerdateien bestehen, die direkt verwendet und in Projekten eingebunden werden können, müssen einige Bibliotheken kompiliert werden. Um die Installation der Boost-Bibliotheken für Entwickler so einfach wie möglich zu gestalten, steht mit Boost.Build ein Werkzeug für einen automatisierten Installationsprozess zur Verfügung. Anstatt Bibliotheken einzeln zu überprüfen und dann gegebenenfalls zu kompilieren, kann mit Boost.Build das Gesamtpaket automatisch installiert werden. Dabei werden von Haus aus zahlreiche Betriebssysteme und Compiler unterstützt, so dass Sie keine langwierigen Einstellungen in Konfigurationsdateien vornehmen müssen.
Um mit Hilfe von Boost.Build die Boost-Bibliotheken automatisch zu installieren, wird auf ein Programm namens **bjam** zugegriffen. Dieses Programm wird seinerseits in Form von Quellcode zur Verfügung gestellt und muss daher erst kompiliert werden. Das ist der Grund, warum der Installationsprozess aus zwei Schritten besteht. Wechseln Sie in Ihr Boost-Verzeichnis, nachdem Sie die Boost-Bibliotheken heruntergeladen und entpackt haben, und geben Sie folgende zwei Befehle nacheinander auf der Kommandozeile ein:

1. Geben Sie unter Windows **`bootstrap`** und unter anderen Betriebssystemen wie Linux **`./bootstrap.sh`** ein, um **bjam** zu kompilieren. Es wird automatisch ein auf Ihrem System installierter C-Compiler gesucht, um **bjam** zu erstellen.

2. Geben Sie anschließend unter Windows **`bjam`** und unter anderen Betriebssystemen wie Linux **`./bjam`** ein, um die Installation zu starten.

bootstrap verwenden Sie nur ein einziges Mal, um das Programm **bjam** zu erstellen. Auf **bjam** greifen Sie unter Umständen öfter zu. So können Sie zahlreiche Kommandozeilenparameter angeben, um **bjam** mitzuteilen, wie die Boost-Bibliotheken installiert werden sollen. Wenn Sie **bjam** ohne Kommandozeilenparameter starten, werden Standardeinstellungen verwendet, die nicht immer empfehlenswert sind. Daher sollten Sie die wichtigsten Optionen von **bjam** kennen:

- Mit der Angabe von `stage` oder `install` wählen Sie aus, ob die Boost-Bibliotheken in ein Unterverzeichnis `stage` oder systemweit installiert werden sollen. Was systemweit bedeutet, hängt vom Betriebssystem ab. Unter Windows ist das Zielverzeichnis `C:\Boost`, unter Linux `/usr/local`. Sie können das Zielverzeichnis bei Angabe von `install` über die Option `--prefix` explizit einstellen. Ein Aufruf von **bjam** ohne Kommandozeilenparameter bedeutet automatisch `stage`.

- Wenn Sie **bjam** ohne Angabe von Kommandozeilenparametern aufrufen, wird automatisch nach einem Compiler gesucht. Sie können einen Compiler auswählen, indem Sie ein Toolset über die Option `--toolset` angeben. So können Sie unter Windows Visual C++ 2013 auswählen, indem Sie **bjam** mit `--toolset=msvc-12.0` aufrufen. Unter Linux wählen Sie GCC mit `--toolset=gcc` aus.

- Über den Kommandozeilenparameter `--build-type` wird angegeben, welche Versionen der Bibliotheken erstellt werden sollen. Standardmäßig ist dieser Kommandozeilenparameter auf `minimal` gesetzt, was bedeutet, dass lediglich Release-Versionen erstellt werden. Das kann vor allem ein Problem für Entwickler

werden, die Debug-Versionen ihrer Projekte mit Visual C++ oder GCC erstellen wollen. Da diese Compiler automatisch versuchen, gegen Debug-Versionen der Boost-Bibliotheken zu linken, werden sie nicht fündig und brechen das Linken mit einer Fehlermeldung ab. Deswegen kann es empfehlenswert sein, die Option `--build-type` explizit auf `complete` zu setzen, um Debug- und Release-Versionen der Boost-Bibliotheken zu erstellen. Die Installation kann in diesem Fall sehr lange dauern, was der Grund ist, warum standardmäßig `minimal` gesetzt ist.

- Boost-Bibliotheken, die kompiliert werden, werden unter Windows in Dateien zur Verfügung gestellt, deren Namen Versionsnummern und verschiedene Kürzel beinhalten. Diese lassen erkennen, ob eine Bibliothek zum Beispiel als Debug- oder Release-Variante vorliegt. `libboost_atomic-vc120-mt-gd-1_57` ist so ein Dateiname. Er deutet daraufhin, dass diese Bibliothek mit Visual C++ 2013 erstellt wurde, aus den Boost-Bibliotheken 1.57.0 stammt und als Debug-Variante vorliegt, die in Multithreaded-Anwendungen verwendet werden kann. Über den Kommandozeilenparameter `--layout` können Sie **bjam** anweisen, andere Dateinamen zu erstellen. Wenn Sie ihn zum Beispiel auf `system` setzen, heißt die entsprechende Datei `libboost_atomic`. Unter Linux ist `system` die Voreinstellung. Möchten Sie unter Linux Dateinamen erhalten, wie sie standardmäßig unter Windows erstellt werden, setzen Sie `--layout` auf `versioned`.

Nachdem Sie die wichtigsten Optionen kennengelernt haben, können Sie, wenn Sie mit Visual C++ 2013 arbeiten, Debug- und Release-Versionen der Boost-Bibliotheken mit folgendem Befehl erstellen und in einem Verzeichnis `D:\Boost` installieren:

```
bjam --toolset=msvc-12.0 --build-type=complete --prefix=D:\Boost install
```

Wenn Sie Linux einsetzen und mit GCC arbeiten, verwenden Sie folgenden Befehl, um die Boost-Bibliotheken im standardmäßig vorgesehenen Systemverzeichnis zu installieren:

```
bjam --toolset=gcc --build-type=complete install
```

Es existieren zahlreiche weitere Kommandozeilenparameter, mit denen gezielt angegeben werden kann, wie die Boost-Bibliotheken zu kompilieren sind. Sehen Sie sich zum Beispiel folgenden Befehl an:

```
bjam --toolset=msvc-12.0 debug release link=static runtime-link=shared ↩
   install
```

Die Angaben `debug` und `release` bewirken, dass sowohl Debug- als auch Release-Versionen der Bibliotheken erstellt werden. Die Angabe `link=static` bedeutet, dass ausschließlich statische Bibliotheken erstellt werden. Mit `runtime-link=shared` wiederum wird angegeben, dass die C++-Runtime-Bibliothek dynamisch gelinkt wird, was in Visual C++ 2013 die Standardeinstellung für C++-Anwendungen ist.

Überblick

Es existieren mehr als 100 Boost-Bibliotheken. In diesem Buch werden Ihnen folgende vorgestellt:

Tabelle 1: Vorgestellte Bibliotheken

Boost-Bibliothek	Standard	Kurzbeschreibung
Boost.Accumulators		Boost.Accumulators bietet Akkumulatoren an, denen Zahlen hinzugefügt werden können, um zum Beispiel den Durchschnittswert oder die Standardabweichung zu erhalten.
Boost.Algorithm		Boost.Algorithm bietet verschiedene Algorithmen an, die die Algorithmen der Standardbibliothek ergänzen.
Boost.Any		Boost.Any stellt mit `boost::any` eine Klasse zur Verfügung, die Daten beliebiger Typen speichern kann.
Boost.Array	TR1, C++11	Boost.Array ermöglicht es, ein herkömmliches Array wie einen echten Container zu behandeln.
Boost.Asio		Mit Boost.Asio können Programme entwickelt werden, die Daten asynchron verarbeiten – zum Beispiel Netzwerkanwendungen.

Tabelle 1: (continued)

Boost-Bibliothek	**Standard**	**Kurzbeschreibung**
Boost.Assign		Boost.Assign bietet Hilfsfunktionen an, um mehrere Werte in einem Container speichern zu können, ohne wiederholt Methoden wie `push_back()` aufrufen zu müssen.
Boost.Atomic	C++11	Boost.Atomic bietet mit `boost::atomic` eine Klasse an, um in Multithreaded-Anwendungen atomare Operationen auf integrale Typen anwenden zu können.
Boost.Bimap		Boost.Bimap stellt eine Klasse `boost::bimap` zur Verfügung, die ähnlich funktioniert wie `std::map`. Der entscheidende Unterschied ist, dass bei `boost::bimap` sowohl nach Schlüsseln als auch nach Werten gesucht werden kann.
Boost.Bind	TR1, C++11	Boost.Bind ist eine Art Adapter, der ermöglicht, Funktionen als Template-Parameter zu übergeben, selbst wenn der Funktionskopf mit dem vom Template erwarteten Parameter inkompatibel ist.
Boost.Chrono	C++11	Boost.Chrono bietet zahlreiche Uhren an, um zum Beispiel die aktuelle Uhrzeit zu erhalten oder die CPU-Zeit zu ermitteln.
Boost.CircularBuffer		Boost.CircularBuffer stellt einen ringförmigen Container mit einer konstanten Speichergröße zur Verfügung.
Boost.CompressedPair		Boost.CompressedPair bietet mit `boost::compressed_pair` eine Datenstruktur an, die `std::pair` ähnelt, jedoch weniger Speicherplatz benötigt, wenn Template-Parameter leere Klassen sind.
Boost.Container		Boost.Container stellt alle Container aus der Standardbibliothek zur Verfügung. Darüber hinaus werden von dieser Bibliothek weitere Container wie `boost::container::slist` angeboten.
Boost.Conversion		Boost.Conversion bietet zwei Cast-Operatoren an, um Down- und Crosscasts durchzuführen.
Boost.Coroutine		Boost.Coroutine macht es möglich, Coroutinen, wie sie in anderen Programmiersprachen häufig mit dem Schlüsselwort `yield` in Verbindung gebracht werden, in C++ zu nutzen.
Boost.DateTime		Boost.DateTime kann verwendet werden, wenn Datumsangaben und Uhrzeiten verarbeitet und flexibel formatiert ein- und ausgegeben werden sollen.
Boost.DynamicBitset		Boost.DynamicBitset stellt eine Datenstruktur zur Verfügung, die grundsätzlich genauso funktioniert wie `std::bitset`, jedoch zur Laufzeit konfiguriert wird.
Boost.EnableIf	C++11	Boost.EnableIf macht es möglich, Funktionen basierend auf Typ-Eigenschaften zu überladen.
Boost.Exception		Boost.Exception ermöglicht es, einer geworfenen Ausnahme zusätzliche Daten hinzuzufügen, um in `catch`-Handlern mehr Informationen zur Verfügung zu haben.
Boost.Filesystem		Boost.Filesystem bietet eine Klasse zum Verarbeiten von Pfadangaben und zahlreiche Funktionen an, um auf Dateien und Verzeichnisse zuzugreifen.
Boost.Flyweight		Boost.Flyweight macht es einfach, das gleichnamige Entwurfsmuster einzusetzen – auf Deutsch Fliegengewicht.
Boost.Foreach		Boost.Foreach stellt ein Makro zur Verfügung, das ähnlich wie die range-basierte `for`-Schleife in C++11 funktioniert.

Tabelle 1: (continued)

Boost-Bibliothek	**Standard**	**Kurzbeschreibung**
Boost.Format		Boost.Format ersetzt `std::printf()` durch eine typsichere und erweiterbare Klasse `boost::format`.
Boost.Function	TR1, C++11	Boost.Function vereinfacht die Definition von Funktionszeigern.
Boost.Fusion		Boost.Fusion erlaubt es, heterogene Container zu erstellen – Container, die Werte unterschiedlicher Typen speichern.
Boost.Graph		Boost.Graph bietet Algorithmen an, um in einem Graphen beispielsweise den kürzesten Weg zwischen zwei Punkten zu finden.
Boost.Heap		Boost.Heap bietet zahlreiche Varianten der aus der Standardbibliothek bekannten Klasse `std::priority_queue` an.
Boost.Integer	C++11	Boost.Integer bietet spezialisierte Typen für Ganzzahlen an, wie sie C-Entwicklern seit dem im Jahr 1999 verabschiedeten Standard C99 zur Verfügung stehen.
Boost.Interprocess		Boost.Interprocess gestattet Prozessen, schnell und effizient über gemeinsam genutzte Speicherbereiche zu kommunizieren.
Boost.Intrusive		Boost.Intrusive stellt Container zur Verfügung, die eine höhere Performance als Container aus der Standardbibliothek bieten können. Sie stellen jedoch besondere Anforderungen an Daten, die in ihnen gespeichert werden sollen.
Boost.IOStreams		Boost.IOStreams bietet Streams und Filter an. So kann ein Stream mit einem Filter verbunden werden, um zum Beispiel Daten, die in den Stream geschrieben werden, zu komprimieren.
Boost.Lambda		Boost.Lambda ermöglicht die Definition anonymer Funktionen – also Funktionen ohne Namen – auch ohne C++11.
Boost.LexicalCast		Boost.LexicalCast bietet einen Cast-Operator, mit dem Zahlen in einen String und zurück umgewandelt werden können.
Boost.Lockfree		Boost.Lockfree bietet thread-safe Container an. Auf Container dieser Bibliothek darf von mehreren Threads aus gleichzeitig zugegriffen werden.
Boost.Log		Boost.Log ist die Logging-Bibliothek in Boost.
Boost.MinMax	C++11	Boost.MinMax stellt einen Algorithmus zur Verfügung, mit dem das kleinste und größte Element in einem Container gefunden werden kann, ohne zwei Funktionsaufrufe – nämlich für `std::min()` und `std::max()` – machen zu müssen.
Boost.MPI		Boost.MPI bietet eine C++-Schnittstelle für den MPI-Standard an.
Boost.MetaStateMachine		Boost.MetaStateMachine macht es möglich, Zustandsautomaten zu entwickeln, wie sie in der UML definiert sind.
Boost.MultiArray		Boost.MultiArray erleichtert den Umgang mit mehrdimensionalen Arrays.
Boost.MultiIndex		Mit Boost.MultiIndex können neue Container definiert werden, die gleichzeitig mehrere Schnittstellen wie beispielsweise die von `std::vector` und `std::map` unterstützen.

Tabelle 1: (continued)

Boost-Bibliothek	**Standard**	**Kurzbeschreibung**
Boost.NumericConversion		Boost.NumericConversion bietet einen Cast-Operator an, der Zahlen zwischen Typen unterschiedlicher Bandbreite sicher konvertiert, ohne dass es einen unbemerkten Überlauf geben kann.
Boost.Operators		Mit Boost.Operators können Operatoren automatisch basierend auf anderen bereits definierten Operatoren definiert werden.
Boost.Optional		Boost.Optional bietet eine Klasse an, mit der optionale Rückgabewerte gekennzeichnet werden können. Funktionen, die nicht immer ein Ergebnis zurückgeben können, müssen dann nicht auf spezielle Werte wie -1 oder einen Nullzeiger zugreifen, um anzugeben, dass kein Ergebnis vorliegt.
Boost.Parameter		Boost.Parameter erlaubt es, Parameter an Funktionen als Name/Wert-Paar zu übergeben, wie es zum Beispiel die Programmiersprache Python ermöglicht.
Boost.Phoenix		Boost.Phoenix macht es möglich, ohne C++11 Lambda-Funktionen zu erstellen. Im Gegensatz zu C++11-Lambda-Funktionen sind die mit dieser Bibliothek erstellten Lambda-Funktion generisch – etwas, was in C++ erst seit C++14 unterstützt wird.
Boost.PointerContainer		Boost.PointerContainer bietet Container an, die für die Verwaltung dynamisch reservierter Objekte optimiert sind.
Boost.Pool		Boost.Pool ist eine Bibliothek zur Verwaltung von Speicher. So bietet Boost.Pool beispielsweise einen Allokator, der Speicher unter Umständen schneller zur Verfügung stellen kann als der Standardallokator.
Boost.ProgramOptions		Boost.ProgramOptions erlaubt es einer Anwendung, Kommandozeilenparameter zu definieren und auszuwerten.
Boost.PropertyTree		Boost.PropertyTree stellt einen Container zur Verfügung, in dem Schlüssel/Wert-Paare in einer Baumstruktur gespeichert werden. Auf diese Weise sollen Konfigurationsdaten, wie sie in vielen Anwendungen verwendet werden, einfach verwaltet werden können.
Boost.Random	TR1, C++11	Boost.Random stellt Generatoren für Zufallszahlen zur Verfügung.
Boost.Range		Boost.Range führt mit der Range ein Konzept ein, das die Iteratoren ersetzt, die üblicherweise von Containern über `begin()` und `end()` erhalten und als Paar an Algorithmen übergeben werden müssen.
Boost.Ref	TR1, C++11	Die Adapter von Boost.Ref ermöglichen es, Referenzen auf Objekte, die nicht kopiert werden können oder sollen, an Funktionen zu übergeben, die Kopien erwarten.
Boost.Regex	TR1, C++11	Boost.Regex bietet Funktionen an, um Strings mit Hilfe regulärer Ausdrücke zu durchsuchen.
Boost.ScopeExit		Boost.ScopeExit bietet Makros an, um Code-Blöcke zu definieren, die ausgeführt werden, wenn der aktuelle Gültigkeitsbereich endet. So kann sichergestellt werden, dass Ressourcen am Ende eines Gültigkeitsbereichs freigegeben werden, ohne auf Smartpointer oder andere Klassen zugreifen zu müssen.

Tabelle 1: (continued)

Boost-Bibliothek	**Standard**	**Kurzbeschreibung**
Boost.Serialization		Mit Boost.Serialization können Objekte serialisiert und in Dateien gespeichert und später wieder von ihnen geladen werden.
Boost.Signals2		Boost.Signals2 ist ein Framework zur Ereignisverarbeitung. Es setzt das Signal-Slot-Konzept um: Funktionen werden mit Signalen verbunden und automatisch aufgerufen, wenn Signale ausgelöst werden.
Boost.SmartPoiners	TR1, C++11	Boost.SmartPointers bietet zahlreiche Smartpointer an, die die Verwaltung von dynamisch reservierten Objekten vereinfachen.
Boost.Spirit		Boost.Spirit ermöglicht es, Parser zu generieren, indem Regeln in einer Syntax niedergeschrieben werden, die der EBNF (Erweiterte Backus-Naur-Form) ähnelt.
Boost.StringAlgorithms		In der Bibliothek Boost.StringAlgorithms stehen zahlreiche freistehende Funktionen zur Verfügung, die es einfacher machen, Strings zu verarbeiten.
Boost.Swap		Boost.Swap bietet mit `boost::swap()` einen Algorithmus an, der sich so verhält wie `std::swap()`, jedoch für viele Boost-Bibliotheken optimiert ist.
Boost.System	C++11	Boost.System stellt ein Framework zur Verfügung, um system- und anwendungsspezifische Fehlercodes zu verarbeiten.
Boost.Thread	C++11	Boost.Thread ermöglicht es, Multithreaded-Programme zu entwickeln.
Boost.Timer		Boost.Timer bietet Uhren an, um die Ausführungsgeschwindigkeit von Code zu messen.
Boost.Tokenizer		Mit Boost.Tokenizer kann über einzelne Bestandteile eines Strings iteriert werden.
Boost.Tribool		Boost.Tribool bietet einen Typ an, der nicht wie bool zwei, sondern drei Status kennt.
Boost.Tuple	TR1, C++11	Boost.Tuple bietet eine verallgemeinerte Version von `std::pair` an. So können in der von Boost.Tuple zur Verfügung gestellten Klasse nicht nur zwei, sondern beliebig viele Werte gruppiert werden.
Boost.TypeTraits	TR1, C++11	Boost.TypeTraits bietet Funktionen an, um Typen auf Eigenschaften zu überprüfen.
Boost.Unordered	TR1, C++11	Boost.Unordered bietet mit `boost::unordered_set` und `boost::unordered_map` zwei Hash-Container an.
Boost.Utility		Boost.Utility ist ein Sammelsurium verschiedener Hilfsmittel, die jeweils zu klein sind, um in ihrer eigenen Bibliothek gepflegt zu werden. Was keine eigene Bibliothek rechtfertigt und nicht in andere Bibliotheken passt, wird in Boost.Utility abgelegt.
Boost.Uuid		Boost.Uuid bietet eine Klasse `boost::uuids::uuid` und Generatoren zur Erzeugung von UUIDs an.
Boost.Variant		Boost.Variant ermöglicht die Definition von Typen, die ähnlich wie `union` mehrere Typen gruppieren. Der Vorteil von Boost.Variant ist, dass in dieser Gruppe beliebige Klassen verwendet werden können.
Boost.Xpressive		Boost.Xpressive ermöglicht es, reguläre Ausdrücke auf Strings anzuwenden. Reguläre Ausdrücke werden nicht als Strings angegeben, sondern als C++-Code.

Die voraussichtlich nächste Version des Standards wird C++17 sein. Rund um C++17 existieren zahlreiche auf bestimmte Themenblöcke spezialisierte Projektgruppen. Diese sind als Technische Spezifikationen bekannt – kurz TS. In der File System TS wird zum Beispiel basierend auf Boost.Filesystem über eine Erweiterung des Standards zum Zugriff auf Dateien und Verzeichnisse nachgedacht. Weitere Informationen zu C++17 und dem Standardisierungsprozess zu C++ finden Sie unter isocpp.org.

Teil I

RAII und Speicherverwaltung

RAII steht für resource acquisition is initialization, was in etwa bedeutet, dass das Öffnen von Ressourcen mit der Initialisierung von Objekten verknüpft werden soll. Smartpointer sind ein prominentes Beispiel für RAII. Sie helfen, Speicherlecks zu vermeiden. Die folgenden Bibliotheken stellen Smartpointer und andere Hilfmittel zur Verfügung, um Speicher einfacher zu verwalten.

- Boost.SmartPointers definiert zahlreiche Smartpointer. Einige dieser Smartpointer werden auch von der C++11-Standardbibliothek bereitgestellt. Andere finden sich ausschließlich in Boost.
- Boost.PointerContainer stellt Container zur Verfügung, in denen dynamisch allokierte – also mit `new` erstellte – Objekte gespeichert werden können. Die von dieser Bibliothek angebotenen Container geben die Objekte im Destruktur mit `delete` frei, so dass Sie keine Smartpointer verwenden müssen.
- Boost.Smartpointers und Boost.PointerContainer funktionieren nur mit Zeigern auf dynamisch allokierte Objekte. Möchten Sie andere Ressourcen zuverlässig am Ende eines Gültigkeitsbereichs schließen, können Sie entweder ressourcen-spezifische Klassen verwenden oder greifen auf die Bibliothek Boost.ScopeExit zu.
- Boost.Pool hat nichts mit RAII, aber viel mit Speicherverwaltung zu tun. Diese Bibliothek bietet zahlreiche Klassen an, um Speicher in Ihrem Programm schneller zur Verfügung zu stellen.

Kapitel 1

Boost.SmartPointers

Die Bibliothek Boost.SmartPointers bietet verschiedene Smartpointer an. Sie helfen, dynamisch reservierte Objekte zu verwalten. Diese werden in Smartpointern verankert, die die dynamisch reservierten Objekte im Destruktor freigeben. Da Destruktoren ausgeführt werden, wenn der Gültigkeitsbereich von Smartpointern endet, ist die Freigabe von dynamisch reservierten Objekten garantiert. Es kann nicht zu Speicherlecks kommen, weil ein `delete` vergessen wurde.
Die Standardbibliothek kennt seit C++98 den Smartpointer `std::auto_ptr`. Dieser gilt seit C++11 als veraltet. Mit dieser Version haben neue und bessere Smartpointer Einzug in den Standard gehalten. `std::shared_ptr` und `std::weak_ptr` stammen ursprünglich aus Boost.SmartPointers und heißen hier `boost::shared_ptr` und `boost::weak_ptr`. Zu `std::unique_ptr` gibt es kein Gegenstück. Dafür bietet Boost.SmartPointers mit `boost::scoped_ptr`, `boost::scoped_array`, `boost::shared_array` und `boost::intrusive_ptr` vier weitere Smartpointer an, die in der Standardbibliothek fehlen.

> Anmerkung
>
> Die Boost-Bibliotheken bieten seit der Version 1.57.0 ein Gegenstück zu `std::unique_ptr` an. Die Klasse heißt `boost::movelib::unique_ptr` und befindet sich nicht in Boost.SmartPointers, sondern in Boost.Move. Boost.Move wird in diesem Buch nicht vorgestellt. Diese Bibliothek bietet eine Emulation von Rvalue-Referenzen für Entwicklungsumgebungen an, die C++11 nicht unterstützen. Arbeiten Sie mit C++11, können Sie `boost::movelib::unique_ptr` und Boost.Move ignorieren.

1.1 Alleiniges Eigentum

`boost::scoped_ptr` ist ein Smartpointer, der alleiniger Eigentümer eines dynamisch reservierten Objekts ist. `boost::scoped_ptr` kann weder kopiert noch verschoben werden. Der Smartpointer ist in der Headerdatei `boost/scoped_ptr.hpp` definiert.

Beispiel 1.1 `boost::scoped_ptr` in Aktion

```
#include <boost/scoped_ptr.hpp>
#include <iostream>

int main()
{
  boost::scoped_ptr<int> p{new int{1}};
  std::cout << *p << '\n';
  p.reset(new int{2});
  std::cout << *p.get() << '\n';
  p.reset();
  std::cout << std::boolalpha << static_cast<bool>(p) << '\n';
}
```

Ein Smartpointer vom Typ `boost::scoped_ptr` kann das Eigentum an einem Objekt nicht übertragen. Einmal mit einer Adresse initialisiert, wird das dynamisch reservierte Objekte freigegeben, wenn der Destruktor ausgeführt wird oder eine Methode `reset()` aufgerufen wird.
Im Beispiel 1.1 wird ein Smartpointer **p** vom Typ boost::scoped_ptr<int> verwendet. **p** wird mit einem Zeiger auf ein dynamisch reserviertes Objekt initialisiert, das die Zahl 1 speichert. Über den Operator `operator*` wird **p** dereferenziert und `1` auf die Standardausgabe ausgegeben.
Mit `reset()` kann im Smartpointer eine neue Adresse gespeichert werden. So wird im Beispiel die Adresse eines neuen dynamisch reservierten int-Objekts mit der Zahl 2 an **p** übergeben. Beim Aufruf von `reset()` wird automatisch das bisher im Smartpointer referenzierte Objekt mit `delete` zerstört.
`get()` gibt die Adresse des Objekts zurück, auf das der Smartpointer verweist. Im Beispiel wird die Adresse, die von `get()` zurückgegeben wird, dereferenziert und `2` in die Standardausgabe geschrieben.
`boost::scoped_ptr` überlädt den Operator `operator bool`. `operator bool` gibt `true` zurück, wenn der Smartpointer eine Referenz auf ein Objekt besitzt – also nicht leer ist. Das Beispiel gibt entsprechend `false` aus, da **p** durch einen Aufruf von `reset()` zurückgesetzt wurde.
Im Destruktor von `boost::scoped_ptr` wird das referenzierte Objekt mit `delete` freigegeben. Daher dürfen Sie `boost::scoped_ptr` nicht mit der Adresse eines dynamisch reservierten Arrays initialisieren – dieses muss mit `delete[]` freigegeben werden. Für Arrays bietet Boost.SmartPointers die Klasse `boost::scoped_array` an.

Beispiel 1.2 `boost::scoped_array` in Aktion

```
#include <boost/scoped_array.hpp>

int main()
{
  boost::scoped_array<int> p{new int[2]};
  *p.get() = 1;
  p[1] = 2;
  p.reset(new int[3]);
}
```

Der Smartpointer `boost::scoped_array` wird wie `boost::scoped_ptr` verwendet. Der entscheidende Unterschied ist, dass der Destruktor von `boost::scoped_array` dynamisch reservierten Speicher mit `delete[]` freigibt. Da ausschließlich Arrays mit `delete[]` freigegeben werden, müssen Sie `boost::scoped_array` mit der Adresse eines dynamisch reservierten Arrays initialisieren.
`boost::scoped_array` ist in der Headerdatei `boost/scoped_array.hpp` definiert.
`boost::scoped_array` überlädt die beiden Operatoren `operator[]` und `operator bool`. Über `operator[]` können Sie direkt auf eine Position im Array zugreifen – ein Objekt vom Typ `boost::scoped_array` verhält sich wie das Array, dessen Adresse es speichert. So wird im Beispiel 1.2 die Zahl 2 als zweites Element im Array gespeichert, auf das **p** verweist.
Wie bei `boost::scoped_ptr` stehen ebenfalls die beiden Methoden `get()` und `reset()` zur Verfügung, mit denen die im `boost::scoped_array` gespeicherte Adresse abgerufen und neu gesetzt werden kann.

1.2 Geteiltes Eigentum

Der Smartpointer `boost::shared_ptr` ähnelt `boost::scoped_ptr`. Der entscheidende Unterschied ist, dass `boost::shared_ptr` nicht exklusiver Eigentümer eines Objekts ist, sondern mit anderen Smartpointern vom Typ `boost::shared_ptr` das Eigentum teilen kann. Das Objekt, dessen Adresse in mehreren Kopien eines Smartpointers vom Typ `boost::shared_ptr` gespeichert ist, wird erst dann zerstört, wenn die letzte Kopie zerstört wird. Da `boost::shared_ptr` das Eigentum teilen kann, können Kopien dieses Smartpointers erstellt werden – etwas, was mit `boost::scoped_ptr` nicht möglich ist.
`boost::shared_ptr` ist in der Headerdatei `boost/shared_ptr.hpp` definiert.

Beispiel 1.3 `boost::shared_ptr` in Aktion

```
#include <boost/shared_ptr.hpp>
#include <iostream>

int main()
{
  boost::shared_ptr<int> p1{new int{1}};
  std::cout << *p1 << '\n';
```

```
  boost::shared_ptr<int> p2{p1};
  p1.reset(new int{2});
  std::cout << *p1.get() << '\n';
  p1.reset();
  std::cout << std::boolalpha << static_cast<bool>(p2) << '\n';
}
```

Im Beispiel 1.3 werden zwei Smartpointer **p1** und **p2** vom Typ `boost::shared_ptr` verwendet. **p2** wird mit **p1** initialisiert, woraufhin sich beide Smartpointer das Eigentum an der int-Variablen mit dem Wert 1 teilen. Wenn für **p1** `reset()` aufgerufen wird, wird eine neue int-Variable in **p1** verankert. Dies bedeutet nicht, dass die bereits existierende int-Variable zerstört wird. Da diese auch in **p2** verankert ist, existiert sie weiterhin. Nach dem ersten Aufruf von `reset()` ist **p1** alleiniger Eigentümer einer int-Variablen mit dem Wert 2 und **p2** alleiniger Eigentümer einer int-Variablen mit dem Wert 1.
`boost::shared_ptr` verwendet intern einen Referenzzähler. Erst wenn `boost::shared_ptr` feststellt, dass die letzte Kopie eines Smartpointers zerstört wird, wird das entsprechende Objekt mit `delete` freigegeben.
`boost::shared_ptr` überlädt ähnlich wie `boost::scoped_ptr` die Operatoren `operator*()`, `operator->()` und `operator bool()`. Darüberhinaus stehen die Methoden `get()` und `reset()` zur Verfügung, um die gespeicherte Adresse abzurufen und eine neue Adresse zu setzen.
Dem Konstruktor von `boost::shared_ptr` kann als zweiter Parameter ein *Deleter* übergeben werden. Dabei muss es sich um eine Funktion oder ein Funktionsobjekt handeln, das als einzigen Parameter einen Zeiger auf den Typ erwartet, mit dem `boost::shared_ptr` instanziiert ist. Der Deleter wird im Destruktor von `boost::shared_ptr` anstelle von `delete` verwendet. So ist es möglich, andere Ressourcen als dynamisch reservierte Objekte in einem `boost::shared_ptr` zu verwalten.

Beispiel 1.4 `boost::shared_ptr` mit benutzerdefiniertem Deleter

```
#include <boost/shared_ptr.hpp>
#include <Windows.h>

int main()
{
  boost::shared_ptr<void> handle(OpenProcess(PROCESS_SET_INFORMATION, FALSE,
    GetCurrentProcessId()), CloseHandle);
}
```

Im Beispiel 1.4 wird `boost::shared_ptr` mit dem Typ void instanziiert. Als ersten Parameter wird dem Konstruktor der Rückgabewert von `OpenProcess()` übergeben. Es handelt sich hierbei um eine Windows-Funktion, um einen Handle auf einen Prozess zu erhalten. Über den Aufruf von `OpenProcess()` wird im Beispiel ein Handle auf den eigenen Prozess erhalten – auf das Beispiel selber.
Handle stellen unter Windows Verweise auf Ressourcen dar, die geschlossen werden müssen, wenn sie nicht mehr benötigt werden. Dazu stellt Windows die Funktion `CloseHandle()` zur Verfügung. Dieser muss als einziger Parameter der Handle für die Ressource übergeben werden, die geschlossen werden soll. Im Beispiel wird `CloseHandle()` als zweiter Parameter an den Konstruktor von `boost::shared_ptr` übergeben. `CloseHandle()` stellt den Deleter von **handle** dar. Wenn **handle** am Ende von `main()` zerstört wird, greift der Destruktor auf `CloseHandle()` zu und gibt die Ressource frei, deren Handle als erster Parameter an den Konstruktor übergeben wurde.

> **Anmerkung**
>
> Beachten Sie, dass das Beispiel nur deswegen funktioniert, weil ein Handle als void* definiert ist. Würde `OpenProcess()` nicht den Rückgabewert void* haben und `CloseHandle()` keinen Parameter vom Typ void* erwarten, könnte `boost::shared_ptr` nicht mit void instanziiert werden. `boost::shared_ptr` ist kein Allheilmittel, das sich dank Deleter zur Verwaltung beliebiger Ressourcen eignet.

Beispiel 1.5 `boost::make_shared` in Aktion

```
#include <boost/make_shared.hpp>
#include <typeinfo>
#include <iostream>
```

```
int main()
{
  auto p1 = boost::make_shared<int>(1);
  std::cout << typeid(p1).name() << '\n';
  auto p2 = boost::make_shared<int[]>(10);
  std::cout << typeid(p2).name() << '\n';
}
```

Boost.SmartPointers bietet in der Headerdatei `boost/make_shared.hpp` eine Hilfsfunktion `boost::make_shared()` an. Sie können mit `boost::make_shared()` Smartpointer vom Typ `boost::shared_ptr` erstellen, ohne auf diese Klasse direkt zugreifen und den Konstruktor aufrufen zu müssen.
`boost::make_shared()` hat den Vorteil, dass der Speicher für das dynamisch zu reservierende Objekt und für den vom Smartpointer verwendeten Referenzzähler in einem Block reserviert werden kann. `boost::make_shared()` ist effizienter, als wenn Sie das dynamisch zu reservierende Objekt mit `new` erstellen und der Konstruktor von `boost::shared_ptr` ein weiteres Mal auf `new` zugreift, um Speicher für den Referenzzähler zu reservieren.
Sie können `boost::make_shared()` auch für Arrays verwenden. So führt der zweite Aufruf von `boost::make_shared()` im Beispiel 1.5 dazu, dass in **`p2`** ein int-Array mit zehn Elementen verankert wird.
Beachten Sie, dass es erst seit Boost 1.53.0 möglich ist, `boost::shared_ptr` für Arrays zu verwenden. Boost.SmartPointers bietet mit `boost::shared_array` einen Smartpointer an, der sich zu `boost::shared_ptr` verhält wie `boost::scoped_array` zu `boost::scoped_ptr`. Wenn Sie obiges Beispiel mit Visual C++ 2013 und Boost 1.53.0 oder neuer ausführen, wird für **`p2`** `class boost::shared_ptr<int [0]>` ausgegeben. Seit Boost 1.53.0 unterstützt `boost::shared_ptr` sowohl einzelne Objekte als auch Arrays und erkennt, ob die Freigabe im Destruktor mit `delete` oder `delete[]` erfolgen muss. Da `boost::shared_ptr` seit Boost 1.53.0 außerdem den Operator `operator[]` anbietet, stellt dieser Smartpointer eine Alternative zu `boost::shared_array` dar.

Beispiel 1.6 `boost::shared_array` in Aktion

```
#include <boost/shared_array.hpp>
#include <iostream>

int main()
{
  boost::shared_array<int> p1{new int[1]};
  {
    boost::shared_array<int> p2{p1};
    p2[0] = 1;
  }
  std::cout << p1[0] << '\n';
}
```

`boost::shared_array` ergänzt `boost::shared_ptr`: Da `boost::shared_array` im Destruktor `delete[]` ausführt, kann dieser Smartpointer für dynamisch reservierte Arrays verwendet werden. Vor Boost 1.53.0 musste `boost::shared_array` für Arrays verwendet werden – `boost::shared_ptr` unterstützte Arrays nicht.
`boost::shared_array` ist in der Headerdatei `boost/shared_array.hpp` definiert.
Im Beispiel 1.6 teilen sich die Smartpointer **`p1`** und **`p2`** das Eigentum an einem dynamisch reservierten int-Array. Wenn über `operator[]` auf **`p2`** zugegriffen wird, um den Wert 1 im Array zu speichern, kann über **`p1`** auf das gleiche dynamisch reservierte Array zugegriffen werden. Das Beispiel gibt entsprechend `1` auf die Standardausgabe aus.
`boost::shared_array` verwendet wie `boost::shared_ptr` einen Referenzzähler. Das dynamisch reservierte Array wird nicht freigegeben, wenn der Gültigkeitsbereich von **`p2`** endet. Zu diesem Zeitpunkt existiert noch eine Referenz von **`p1`** auf das Array. Erst am Ende von `main()`, wenn der Gültigkeitsbereich von **`p1`** endet, wird das Array zerstört.
`boost::shared_array` bietet ebenfalls die Methoden `get()` und `reset()` an. Außerdem ist der Operator `operator bool` überladen.

Beispiel 1.7 `boost::shared_ptr` mit `BOOST_SP_USE_QUICK_ALLOCATOR`

```
#define BOOST_SP_USE_QUICK_ALLOCATOR
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <ctime>
```

```
int main()
{
  boost::shared_ptr<int> p;
  std::time_t then = std::time(nullptr);
  for (int i = 0; i < 1000000; ++i)
    p.reset(new int{i});
  std::time_t now = std::time(nullptr);
  std::cout << now - then << '\n';
}
```

Beachten Sie, dass es von Vorteil sein kann, Smartpointer wie `boost::shared_ptr` denen aus der Standardbibliothek vorzuziehen. Boost.SmartPointers bietet Makros an, um das Verhalten der Smartpointer zu optimieren. So wird im Beispiel 1.7 das Makro `BOOST_SP_USE_QUICK_ALLOCATOR` verwendet, um einen in Boost.SmartPointers integrierten Allokator zu aktivieren. Der Allokator reserviert und verwaltet Speicherblöcke, um die Anzahl der Aufrufe von `new` und `delete` zu minimieren, die zur Allokation von Referenzzählern benötigt werden. Im Beispiel wird mit `std::time()` die Zeit vor und nach der Schleife gemessen. Während die gemessene Ausführungsgeschwindigkeit vom verwendeten Computer abhängt, kann das Programm mit `BOOST_SP_USE_QUICK_ALLOCATOR` schneller sein als ohne. `BOOST_SP_USE_QUICK_ALLOCATOR` wird in der Dokumentation der Boost-Bibliothek nicht erwähnt. Sie sollten daher die Ausführungsgeschwindigkeit Ihres Programms ausführlich messen und die Ergebnisse vergleichen, die Sie mit und ohne `BOOST_SP_USE_QUICK_ALLOCATOR` erhalten.

> **Tipp**
>
> Neben `BOOST_SP_USE_QUICK_ALLOCATOR` unterstützt Boost.SmartPointers weitere Makros wie zum Beispiel `BOOST_SP_ENABLE_DEBUG_HOOKS`. Die Namen der Makros beginnen mit BOOST_SP_, so dass eine Suche in den Headerdateien von Boost.SmartPointers einen raschen Überblick über die unterstützten Makros verschafft.

1.3 Spezielle Smartpointer

Alle in diesem Kapitel kennengelernten Smartpointer können für sich allein genommen in unterschiedlichen Situationen verwendet werden. Der Smartpointer vom Typ `boost::weak_ptr` ergibt jedoch nur im Zusammenspiel mit `boost::shared_ptr` Sinn. Er ist in der Headerdatei `boost/weak_ptr.hpp` definiert.

Beispiel 1.8 `boost::weak_ptr` in Aktion

```
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
#include <thread>
#include <functional>
#include <iostream>

void reset(boost::shared_ptr<int> &sh)
{
  sh.reset();
}

void print(boost::weak_ptr<int> &w)
{
  boost::shared_ptr<int> sh = w.lock();
  if (sh)
    std::cout << *sh << '\n';
}

int main()
{
  boost::shared_ptr<int> sh{new int{99}};
  boost::weak_ptr<int> w{sh};
```

```
  std::thread t1{reset, std::ref(sh)};
  std::thread t2{print, std::ref(w)};
  t1.join();
  t2.join();
}
```

Die Klasse `boost::weak_ptr` muss mit einem `boost::shared_ptr` initialisiert werden. Die wichtigste von ihr angebotene Methode ist `lock()`. `lock()` gibt einen `boost::shared_ptr` zurück, der das Eigentum mit dem Smartpointer teilt, mit dem `boost::weak_ptr` initialisiert wurde. Für den Fall, dass der `boost::shared_ptr`, der bei der Initialisierung angegeben wurde, nicht mehr auf ein Objekt zeigt, ist der von `lock()` zurückgegebene `boost::shared_ptr` leer.
`boost::weak_ptr` bietet sich an, wenn eine Funktion mit einem in einem `boost::shared_ptr` verwalteten Objekt arbeiten soll, die Lebensdauer dieses Objekts jedoch nicht von dieser Funktion abhängen soll. Die Funktion soll nur dann mit dem Objekt arbeiten, wenn es momentan in Smartpointern vom Typ `boost::shared_ptr`, die an anderen Stellen im Programm existieren, verankert ist. Verschwinden diese Smartpointer, soll das entsprechende Objekt nicht mehr existieren und nicht unnötigerweise durch einen zusätzlichen Smartpointer vom Typ `boost::shared_ptr` in der entsprechenden Funktion am Leben gehalten werden.
Im Beispiel 1.8 werden in `main()` zwei Threads erstellt. Ein Thread führt die Funktion `reset()` aus, die eine Referenz auf einen `boost::shared_ptr` erhält. Der andere Thread besteht aus der Funktion `print()`, die eine Referenz auf einen `boost::weak_ptr` erhält. Dieser `boost::weak_ptr` wird in der Funktion `main()` mit dem `boost::shared_ptr` initialisiert.
Wenn das Programm gestartet wird, werden die beiden Funktionen `reset()` und `print()` gleichzeitig in zwei Threads ausgeführt. Es lässt sich nicht vorhersagen, in welcher Reihenfolge die verschiedenen Anweisungen in den Threads abgearbeitet werden. Das Problem ist, dass in der Funktion `reset()` ein Objekt zerstört werden soll, das gleichzeitig in einem anderen Thread verarbeitet und auf die Standardausgabe ausgegeben werden soll. Es könnte sein, dass in der Funktion `reset()` das Objekt zerstört wird, während in der Funktion `print()` auf dieses Objekt zugegriffen wird, um es auszugeben.
`boost::weak_ptr` löst diese Problematik wie folgt: Beim Aufruf von `lock()` wird ein `boost::shared_ptr` zurückgeben. Dieser `boost::shared_ptr` zeigt auf ein Objekt, wenn dieses Objekt zum Zeitpunkt des Aufrufs existiert. Andernfalls ist der `boost::shared_ptr` leer und ein Null-Zeiger.
`boost::weak_ptr` allein hat keinen Einfluss auf die Lebensdauer eines Objekts. Damit in `print()` gefahrlos auf das Objekt zugegriffen kann, ohne dass es in einem anderen Thread zerstört wird, gibt `lock()` einen `boost::shared_ptr` zurück. Wird in einem anderen Thread versucht, das Objekt zu zerstören, existiert es dank des von `lock()` zurückgegebenen Smartpointers vom Typ `boost::shared_ptr` weiter.

Beispiel 1.9 `boost::intrusive_ptr` in Aktion

```
#include <boost/intrusive_ptr.hpp>
#include <atlbase.h>
#include <iostream>

void intrusive_ptr_add_ref(IDispatch *p) { p->AddRef(); }
void intrusive_ptr_release(IDispatch *p) { p->Release(); }

void check_windows_folder()
{
  CLSID clsid;
  CLSIDFromProgID(CComBSTR{"Scripting.FileSystemObject"}, &clsid);
  void *p;
  CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER, __uuidof(IDispatch), &p);
  boost::intrusive_ptr<IDispatch> disp{static_cast<IDispatch*>(p), false};
  CComDispatchDriver dd{disp.get()};
  CComVariant arg{"C:\\Windows"};
  CComVariant ret{false};
  dd.Invoke1(CComBSTR{"FolderExists"}, &arg, &ret);
  std::cout << std::boolalpha << (ret.boolVal != 0) << '\n';
}

int main()
{
  CoInitialize(0);
  check_windows_folder();
  CoUninitialize();
```

```
}
```

Der Smartpointer `boost::intrusive_ptr` funktioniert genauso wie `boost::shared_ptr`. Während `boost::shared_ptr` jedoch automatisch mitzählt, wie viele Kopien momentan auf ein Objekt verweisen und sich das Eigentum teilen, müssen bei `boost::intrusive_ptr` Sie mitzählen. Das ist zum Beispiel von Vorteil, wenn auf Klassen aus Frameworks zugegriffen wird, die sowieso schon mitzählen.
`boost::intrusive_ptr` ist in der Headerdatei `boost/intrusive_ptr.hpp` definiert.
Im Beispiel 1.9 wird auf Microsoft COM-Funktionen zugegriffen – das Beispiel kann ausschließlich unter Windows ausgeführt werden. COM-Objekte bieten sich als Beispiel für `boost::intrusive_ptr` an, da sie mitzählen, wie viele Zeiger momentan auf sie verweisen. Der interne Referenzzähler eines COM-Objekts kann über die Methoden `AddRef()` und `Release()` jeweils um 1 erhöht und verringert werden. Wird der interne Zähler nach einem Aufruf von `Release()` auf 0 gesetzt, wird das COM-Objekt automatisch zerstört.
Die beiden Methoden `AddRef()` und `Release()` werden innerhalb der Funktionen `intrusive_ptr_add_ref()` und `intrusive_ptr_release()` aufgerufen, um den internen Zähler im COM-Objekt zu inkrementieren und dekrementieren. Boost.SmartPointers erwartet, dass Sie diese beiden Funktionen zur Verfügung stellen. Sie werden automatisch aufgerufen, wenn der interne Zähler inkrementiert oder dekrementiert werden muss. Der Parameter, der an diese Funktionen übergeben wird, ist ein Zeiger auf den Typ, mit dem das Template `boost::intrusive_ptr` instanziiert wurde.
Das COM-Objekt, das im Beispiel 1.9 verwendet wird, heißt FileSystemObject und ist standardmäßig unter Windows vorhanden. Es gestattet einen Zugriff aufs Dateisystem, um zum Beispiel zu überprüfen, ob ein bestimmtes Verzeichnis existiert. Im Beispiel wird überprüft, ob es das Verzeichnis `C:\Windows` gibt. Wie dies im Detail funktioniert, hängt von COM ab und ist unwichtig, um die Funktionsweise von `boost::intrusive_ptr` zu verstehen. Entscheidend ist, dass am Ende der Funktion `check_windows_folder()`, wenn der Gültigkeitsbereich von **`disp`** endet, die Funktion `intrusive_ptr_release()` automatisch aufgerufen wird. Nachdem dort mit `Release()` der interne Zähler auf 0 gesetzt wird, wird das COM-Objekt FileSystemObject zerstört.
Der Parameter `false`, der an den Konstruktor von `boost::intrusive_ptr` übergeben wird, verhindert, dass `intrusive_ptr_add_ref()` aufgerufen wird. Da ein COM-Objekt, wenn es mit `CoCreateInstance()` erstellt wird, einen bereits auf 1 gesetzten Referenzzähler hat, darf dieser nicht noch einmal in `intrusive_ptr_add_ref()` inkrementiert werden.

Kapitel 2

Boost.PointerContainer

Die Bibliothek Boost.PointerContainer bietet für die Verwaltung dynamisch reservierter Objekte spezielle Container an. Verwenden Sie C++11, können Sie einen entsprechenden Container zum Beispiel mit std::vector<std::unique_ptr<int> selbst erstellen. Die von Boost.PointerContainer angebotenen Container können jedoch zusätzlichen Komfort bieten.

Beispiel 2.1 `boost::ptr_vector` in Aktion

```
#include <boost/ptr_container/ptr_vector.hpp>
#include <iostream>

int main()
{
  boost::ptr_vector<int> v;
  v.push_back(new int{1});
  v.push_back(new int{2});
  std::cout << v.back() << '\n';
}
```

Die Klasse `boost::ptr_vector`, die im Beispiel 2.1 verwendet wird und in der Headerdatei `boost/ptr_container/ptr_vector.hpp` definiert ist, funktioniert grundsätzlich genauso wie ein Container vom Typ std::vector<std::unique_ptr<int>>. Da `boost::ptr_vector` jedoch weiß, dass es dynamisch reservierte Objekte speichert, geben Methoden wie `back()` eine Referenz auf ein dynamisch reserviertes Objekt zurück und keinen Zeiger. Das Beispiel gibt entsprechend `2` auf die Standardausgabe aus.

Beispiel 2.2 `boost::ptr_set` mit intuitiv richtiger Sortierung

```
#include <boost/ptr_container/ptr_set.hpp>
#include <boost/ptr_container/indirect_fun.hpp>
#include <set>
#include <memory>
#include <functional>
#include <iostream>

int main()
{
  boost::ptr_set<int> s;
  s.insert(new int{2});
  s.insert(new int{1});
  std::cout << *s.begin() << '\n';

  std::set<std::unique_ptr<int>, boost::indirect_fun<std::less<int>>> v;
  v.insert(std::unique_ptr<int>(new int{2}));
  v.insert(std::unique_ptr<int>(new int{1}));
  std::cout << **v.begin() << '\n';
}
```

Beispiel 2.2 zeigt einen anderen Grund, warum es von Vorteil sein kann, einen spezialisierten Container zu verwenden. In diesem Beispiel werden dynamisch reservierte Variablen vom Typ int in einem `boost::ptr_set` und in einem `std::set` gespeichert. Für das `std::set` wird auf `std::unique_ptr` zugegriffen.

`boost::ptr_set` versteht automatisch, dass die Reihenfolge der Elemente von den gespeicherten int-Werten abhängt. `std::set` hingegen würde `std::unique_ptr` miteinander vergleichen – also Smartpointer und nicht die Variablen, auf die die Smartpointer zeigen. Damit die Sortierung im Beispiel 2.2 auch bei `std::set` auf den int-Werten basiert, muss angegeben werden, wie Vergleiche durchzuführen sind. Hierzu wird auf eine Klasse `boost::indirect_fun` zugegriffen, die aus Boost.PointerContainer stammt. Mit Hilfe dieser Klasse wird `std::set` mitgeteilt, dass die Sortierung nicht basierend auf den Smartpointern durchgeführt werden soll, sondern basierend auf den int-Variablen, auf die die Smartpointer zeigen. Wenn Sie das Beispiel ausführen, wird zweimal `1` ausgegeben.
Neben `boost::ptr_vector` und `boost::ptr_set` stehen weitere Container zur Verfügung, die auf die Verwaltung dynamisch reservierter Objekte spezialisiert sind. Dazu gehören `boost::ptr_deque`, `boost::ptr_list`, `boost::ptr_map`, `boost::ptr_unordered_set` und `boost::ptr_unordered_map`. Diese Container entsprechen den aus der Standardbibliothek bekannten Containern.

Beispiel 2.3 Inserter für Container von Boost.PointerContainer

```
#include <boost/ptr_container/ptr_vector.hpp>
#include <boost/ptr_container/ptr_inserter.hpp>
#include <array>
#include <algorithm>
#include <iostream>

int main()
{
  boost::ptr_vector<int> v;
  std::array<int, 3> a{{0, 1, 2}};
  std::copy(a.begin(), a.end(), boost::ptr_container::ptr_back_inserter(v));
  std::cout << v.size() << '\n';
}
```

Boost.PointerContainer bietet für seine Container Inserter an. Diese befinden sich im Namensraum `boost::ptr_container` und sind in der Headerdatei `boost/ptr_container/ptr_inserter.hpp` definiert. Im Beispiel 2.3 wird auf die Funktion `boost::ptr_container::ptr_back_inserter()` zugegriffen, die einen Inserter vom Typ `boost::ptr_container::ptr_back_insert_iterator` erstellt. Dieser wird an `std::copy()` übergeben, um die Zahlen im Array **a** in den Vektor **v** zu kopieren. Da **v** ein Container vom Typ `boost::ptr_vector` ist, der Adressen von dynamisch reservierten int-Objekten erwartet, muss ein von Boost.PointerContainer angebotener Inserter verwendet werden. Diese erstellen eine Kopie mit `new` auf dem Heap und fügen die Adresse dem Container hinzu.
Neben `boost::ptr_container::ptr_back_inserter()` bietet Boost.PointerContainer die Funktionen `boost::ptr_container::ptr_front_inserter()` und `boost::ptr_container::ptr_inserter()`, um entsprechende Inserter zu erstellen.

Kapitel 3

Boost.ScopeExit

Die Bibliothek Boost.ScopeExit macht es möglich, RAII ohne ressourcenspezifische Klassen zu verwenden.

Beispiel 3.1 `BOOST_SCOPE_EXIT` in Aktion

```
#include <boost/scope_exit.hpp>
#include <iostream>

int *foo()
{
  int *i = new int{10};
  BOOST_SCOPE_EXIT(&i)
  {
    delete i;
    i = 0;
  } BOOST_SCOPE_EXIT_END
  std::cout << *i << '\n';
  return i;
}

int main()
{
  int *j = foo();
  std::cout << j << '\n';
}
```

Boost.ScopeExit bietet mit `BOOST_SCOPE_EXIT` ein Makro an, mit dem eine Art lokale Funktion definiert werden kann. Die Funktion ist ohne Namen. Sie besteht aus einer Parameterliste, die in runden Klammen angegeben wird, und einem Block, der in geschweiften Klammern steht.
Sie müssen die Headerdatei `boost/scoped_exit.hpp` einbinden, um das Makro `BOOST_SCOPE_EXIT` verwenden zu können.
Über die Parameterliste geben Sie Variablen an, auf die Sie im Block zugreifen möchten. Wie bei echten Funktionen findet die Parameterübergabe per Kopie statt. Möchten Sie eine Variable als Referenz übergeben, setzen Sie wie im Beispiel 3.1 ein Ampersand vor den Variablennamen.
Im Block können Sie ausschließlich auf Variablen zugreifen, die in der Parameterliste angegeben sind.
Sinn und Zweck von `BOOST_SCOPE_EXIT` ist es, Code-Blöcke zu definieren, die ausgeführt werden, wenn der Gültigkeitsbereich, in dem sie definiert sind, endet. So wird im Beispiel 3.1 der Block hinter `BOOST_SCOPE_EXIT` ausgeführt, wenn die Funktion `foo()` zurückkehrt.
`BOOST_SCOPE_EXIT` bietet sich für RAII an, ohne ressourcenspezifische Klassen verwenden zu müssen. `foo()` greift auf `new` zu, um dynamisch ein int zu reservieren. Um den Speicher freizugeben, wird mit `BOOST_SCOPE_EXIT` ein entsprechender Block definiert, in dem `delete` zur Anwendung kommt. Der Block wird garantiert ausgeführt – auch dann, wenn die Funktion zum Beispiel vorzeitig aufgrund einer Ausnahme beendet wird. `BOOST_SCOPE_EXIT` steht in diesem Beispiel einem Smartpointer in nichts nach.
Beachten Sie, dass die Variable **`i`** im Block hinter `BOOST_SCOPE_EXIT` auf 0 gesetzt wird. **`i`** wird anschließend von `foo()` zurückgegeben und innerhalb von `main()` auf die Standardausgabe ausgegeben.
Wenn Sie das Beispiel ausführen, sehen Sie, dass nicht 0 ausgegeben wird. **`j`** ist auf einen zufälligen Wert gesetzt – nämlich die Adresse, an der die int-Variable reserviert war. Der Block hinter `BOOST_SCOPE_EXIT` hat **`i`** als Referenz erhalten und den reservierten Speicher freigegeben. Da der Block am Ende von `foo()` ausgeführt wird,

kommt der Schreibzugriff auf **i** zu spät. Der Rückgabewert von `foo()` wurde bereits als Kopie von **i** erstellt, wenn **i** auf 0 gesetzt wird.
Sie können Boost.ScopeExit ignorieren, wenn Sie in einer Entwicklungsumgebung arbeiten, die C++11 unterstützt. Sie können dann RAII ohne ressourcenspezifische Klassen mit Hilfe von Lambda-Funktionen einsetzen.

Beispiel 3.2 Boost.ScopeExit mit C++11-Lambda-Funktionen

```
#include <iostream>
#include <utility>

template <typename T>
struct scope_exit
{
  scope_exit(T &&t) : t_{std::move(t)} {}
  ~scope_exit() { t_(); }
  T t_;
};

template <typename T>
scope_exit<T> make_scope_exit(T &&t) { return scope_exit<T>{
  std::move(t)}; }

int *foo()
{
  int *i = new int{10};
  auto cleanup = make_scope_exit([&i]() mutable { delete i; i = 0; });
  std::cout << *i << '\n';
  return i;
}

int main()
{
  int *j = foo();
  std::cout << j << '\n';
}
```

Beispiel 3.2 definiert eine Klasse `scope_exit`, der im Konstruktor eine Lambda-Funktion übergeben werden kann. Die Lambda-Funktion wird im Destruktor aufgerufen und ausgeführt. Außerdem ist eine Hilfsfunktion `make_scope_exit()` definiert, die es ermöglicht, `scope_exit` zu instanziieren, ohne explizit einen Template-Parameter angeben zu müssen.
In der Funktion `foo()` wird auf `make_scope_exit()` zugegriffen und eine Lambda-Funktion als Parameter übergeben. Die Lambda-Funktion sieht genauso aus wie der Block, der im Beispiel 3.1 hinter `BOOST_SCOPE_EXIT` angegeben wurde: Der Speicherbereich, auf den **i** zeigt, wird mit `delete` freigegeben. Anschließend wird **i** auf 0 gesetzt.
Wenn Sie das Beispiel ausführen, geschieht das Gleiche wie zuvor. So wird nicht nur die dynamisch reservierte int-Variable freigegeben. **j** ist außerdem nicht auf 0 gesetzt, wenn die Variable ausgegeben wird.

Beispiel 3.3 Besonderheiten rund um `BOOST_SCOPE_EXIT`

```
#include <boost/scope_exit.hpp>
#include <iostream>

struct x
{
  int i;

  void foo()
  {
    i = 10;
    BOOST_SCOPE_EXIT(void)
    {
      std::cout << "last\n";
    } BOOST_SCOPE_EXIT_END
    BOOST_SCOPE_EXIT(this_)
    {
      this_->i = 20;
```

```
      std::cout << "first\n";
    } BOOST_SCOPE_EXIT_END
  }
};

int main()
{
  x obj;
  obj.foo();
  std::cout << obj.i << '\n';
}
```

Beispiel 3.3 stellt einige Besonderheiten von `BOOST_SCOPE_EXIT` vor:

- Werden mehrere Blöcke im gleichen Gültigkeitsbereich mit `BOOST_SCOPE_EXIT` definiert, werden sie von hinten nach vorne ausgeführt. Beispiel 3.3 gibt zuerst `first` und dann `last` aus.
- Sollen keine Variablen einem Block hinter `BOOST_SCOPE_EXIT` übergeben werden, muss `void` in den runden Klammern angegeben werden. Die runden Klammern dürfen nicht leer sein.
- Wird `BOOST_SCOPE_EXIT` in einer Methode verwendet, kann über **`this_`** ein Zeiger auf das aktuelle Objekt übergeben werden. Es muss **`this_`** verwendet werden, nicht `this`.

Beispiel 3.3 gibt `first`, `last` und `20` in dieser Reihenfolge aus.

Kapitel 4

Boost.Pool

Bei Boost.Pool handelt es sich um eine Bibliothek, die einige wenige Klassen zur Speicherverwaltung anbietet. Während in C++-Programmen üblicherweise mit `new` dynamisch Speicher angefordert wird und es von der Implementation der Standardbibliothek und vom Betriebssystem abhängt, wie Speicher im Detail zur Verfügung gestellt wird, können Sie mit Boost.Pool auf die Speicherverwaltung Einfluss nehmen. So können Sie beispielsweise die Speicherverwaltung beschleunigen, damit Ihr Programm schneller benötigten Speicher erhält.
Boost.Pool ändert weder die Funktionsweise von `new` noch die des Betriebssystems. Die von Boost.Pool angebotene Speicherverwaltung funktioniert nur deswegen, weil der verwaltete Speicher zuerst vom Betriebssystem angefordert wird – zum Beispiel mit `new`. Von außen betrachtet besitzt Ihr Programm bereits den Speicher. Innerhalb Ihres Programms benötigen Sie den Speicher jedoch noch nicht, sondern übergeben ihn bis auf Weiteres Boost.Pool zur Verwaltung.
Boost.Pool teilt den von Ihnen übergebenen Speicher in gleichgroße Segmente ein. Wann immer Sie Speicher benötigen und von Boost.Pool anfordern, greift Boost.Pool auf das nächste freie Segment zu und weist Ihnen den benötigten Speicher aus diesem Segment zu. Dieses Segment ist dann verbraucht – völlig unabhängig davon, wie viele Bytes Sie genau aus diesem Segment benötigen.
Diese Art der Einteilung wird als *einfach segmentierter Speicher* bezeichnet – auf Englisch simple segregated storage. Es ist die einzige Speicherverwaltung, die Boost.Pool unterstützt. Sie bietet sich vor allem dann an, wenn viele gleichgroße Objekte oft erstellt und zerstört werden müssen, da in diesem Fall der benötigte Speicher sehr schnell zur Verfügung gestellt und freigegeben werden kann.
Boost.Pool bietet mit `boost::simple_segregated_storage` eine Klasse an, die segmentierten Speicher erstellen und verwalten kann. Es handelt sich hierbei um eine Low-Level-Klasse, die Sie in Ihren Programmen üblicherweise nicht direkt verwenden. Sie wird im Beispiel 4.1 lediglich vorgestellt, um die Funktionsweise des einfach segmentierten Speichers zu verdeutlichen. Alle anderen Klassen von Boost.Pool basieren intern auf `boost::simple_segregated_storage`.

Beispiel 4.1 Einfach segmentierter Speicher mit `boost::simple_segregated_storage`

```
#include <boost/pool/simple_segregated_storage.hpp>
#include <vector>
#include <cstddef>

int main()
{
  boost::simple_segregated_storage<std::size_t> storage;
  std::vector<char> v(1024);
  storage.add_block(&v.front(), v.size(), 256);

  int *i = static_cast<int*>(storage.malloc());
  *i = 1;

  int *j = static_cast<int*>(storage.malloc_n(1, 512));
  j[10] = 2;

  storage.free(i);
  storage.free_n(j, 1, 512);
}
```

Um `boost::simple_segregated_storage` verwenden zu können, müssen Sie die Headerdatei `boost/pool/simple_segregated_storage.hpp` einbinden. Sie müssen außerdem einen Parameter bei der Instanziierung der Klasse angeben, da es sich bei `boost::simple_segregated_storage` um ein Template handelt. Im Beispiel 4.1 wird std::size_t übergeben. Der Parameter gibt den Typ für Zahlen an, die als Parameter an Methoden von `boost::simple_segregated_storage` übergeben werden, um zum Beispiel die Größe eines Segments zu beschreiben. Die praktische Bedeutung des Template-Parameters ist eher gering.
Interessanter sind die Methoden, die für `boost::simple_segregated_storage` aufgerufen werden. Zuerst wird mit `add_block()` ein Speicherblock bestehend aus 1024 Bytes an **`storage`** übergeben. Diese Bytes werden von **`v`**, einem Vektor vom Typ `std::vector`, zur Verfügung gestellt. Der dritte an `add_block()` übergebene Parameter gibt an, dass der Speicherblock in Segmente von 256 Bytes eingeteilt werden soll. Da der Speicherblock insgesamt 1024 Bytes groß ist, besteht der Speicher, den **`storage`** verwaltet, aus vier Segmenten.
Mit `malloc()` und `malloc_n()` wird zweimal Speicher von **`storage`** angefordert. Während `malloc()` einen Zeiger auf ein freies Segment zurückgibt, gibt `malloc_n()` einen Zeiger auf ein oder mehrere freie Segmente zurück, die zusammenhängend so viele Bytes zur Verfügung stellen wie mit `malloc_n()` angefordert werden. Im Beispiel 4.1 wird mit `malloc_n()` nach einem zusammenhängenden Speicherbereich von 512 Bytes gesucht. Durch den Aufruf von `malloc_n()` werden zwei Segmente verbraucht, da jedes Segment 256 Bytes groß ist. Nach dem Aufruf von `malloc()` und `malloc_n()` gibt es in **`storage`** nur mehr ein freies Segment.
Am Ende des Beispiels werden mit `free()` und `free_n()` alle Segmente an **`storage`** zurückgegeben. Nach dem Aufruf der beiden Methoden sind alle vier Segmente wieder frei und könnten mit `malloc()` oder `malloc_n()` neu angefordert werden.
Sie verwenden `boost::simple_segregated_storage` üblicherweise nicht direkt. Boost.Pool stellt andere Klassen zur Verfügung, die automatisch Speicher anfordern, ohne dass Sie wie im Beispiel 4.1 Speicher selbst anfordern und an `boost::simple_segregated_storage` übergeben müssen.
Im Beispiel 4.2 wird die Klasse `boost::object_pool` verwendet, die in der Headerdatei `boost/pool/object_pool.hpp` definiert ist.

Beispiel 4.2 Segmentierter Speicher mit `boost::object_pool`

```
#include <boost/pool/object_pool.hpp>

int main()
{
  boost::object_pool<int> pool;

  int *i = pool.malloc();
  *i = 1;

  int *j = pool.construct(2);

  pool.destroy(i);
  pool.destroy(j);
}
```

Ein entscheidender Unterschied zu `boost::simple_segregated_storage` ist, dass `boost::object_pool` den Typ der Objekte kennt, die Sie im Speicher ablegen möchten. So ist im Beispiel 4.2 angegeben, dass **`pool`** einen einfach segmentierten Speicher für int-Werte zur Verfügung stellen soll. Der von **`pool`** verwaltete Speicher wird daraufhin in Segmente von der Größe eines ints eingeteilt – also zum Beispiel in 4-Byte-Blöcke.
Ein weiterer entscheidender Unterschied zu `boost::simple_segregated_storage` ist, dass Sie `boost::object_pool` keinen Speicher zur Verfügung stellen müssen. `boost::object_pool` reserviert Speicher automatisch. So wird im Beispiel 4.2 beim Aufruf von `malloc()` ein Speicherblock reserviert, der 32 int-Werte umfasst. `malloc()` gibt einen Zeiger auf das erste dieser 32 Segmente zurück, in das genau ein int passt. Beachten Sie, dass `malloc()` einen Zeiger vom Typ int* zurückgibt. Es ist kein Cast-Operator nötig, wie er im Beispiel 4.1 bei `boost::simple_segregated_storage` angewandt werden musste.
`construct()` ähnelt `malloc()`, initialisiert ein Objekt jedoch über einen entsprechenden Konstruktoraufruf. So zeigt im Beispiel 4.2 **`j`** auf eine int-Variable, in der 2 gespeichert ist.
Beachten Sie, dass **`pool`** den Aufruf von `construct()` aus dem zur Verfügung stehenden 32 int-Werte umfassenden Speicher bedienen kann. Der Aufruf von `construct()` führt im Beispiel 4.2 nicht dazu, dass **`pool`** Speicher vom Betriebssystem anfordert.
Die letzte im Beispiel 4.2 aufgerufene Methode `destroy()` gibt eine int-Variable wieder frei.

Beispiel 4.3 Speicherblockgröße von `boost::object_pool` verändern

```
#include <boost/pool/object_pool.hpp>
```

```
#include <iostream>

int main()
{
  boost::object_pool<int> pool{32, 0};
  pool.construct();
  std::cout << pool.get_next_size() << '\n';
  pool.set_next_size(8);
}
```

Sie können dem Konstruktor von `boost::object_pool` zwei Parameter übergeben. Der erste Parameter gibt die Größe des Speicherblocks an, den `boost::object_pool` reserviert, wenn mit `malloc()` oder `construct()` zum ersten Mal ein Segment angefordert wird. Der zweite Parameter gibt die maximale Größe des zu reservierenden Speicherblocks an.
Wenn `malloc()` oder `construct()` so oft aufgerufen werden, dass alle Segmente im Speicherblock reserviert sind, führt der nächste Aufruf einer dieser beiden Methoden dazu, dass `boost::object_pool` einen neuen Speicherblock reserviert. Dieser ist doppelt so groß wie der vorherige Speicherblock. Für jeden weiteren Speicherblock, den `boost::object_pool` benötigt, wird die Größe verdoppelt. `boost::object_pool` kann beliebig viele Speicherblöcke verwalten. Ihre Größen wachsen jedoch exponentiell. Über den zweiten Parameter des Konstruktors ist es möglich, das exponentielle Wachstum zu begrenzen.
Wird der Standardkonstruktor von `boost::object_pool` aufgerufen, ist dies gleichbedeutend mit dem Konstruktoraufruf im Beispiel 4.3. Der erste Parameter gibt an, dass der erste anzufordernde Speicherblock 32 int-Werte umfassen soll. Der zweite Parameter gibt an, dass es keine maximale Größe gibt. Wird 0 übergeben, kann `boost::object_pool` die Speicherblockgröße beliebig oft verdoppeln.
Der Aufruf von `construct()` im Beispiel 4.3 führt dazu, dass **`pool`** einen Speicherblock in der Größe von 32 int-Werten anfordert. **`pool`** kann nun bis zu 32 Aufrufe von `malloc()` oder `construct()` bedienen, ohne erneut Speicher anfordern zu müssen. Müsste erneut Speicher angefordert werden, wäre der nächste Block 64 int-Werte groß.
Über `get_next_size()` kann die Größe des nächsten Speicherblocks erhalten werden, über `set_next_size()` neu gesetzt werden. So gibt im Beispiel 4.3 `get_next_size()` 64 zurück. Der Aufruf von `set_next_size()` bewirkt, dass der nächste Speicherblock, den **`pool`** anfordern würde, keine 64, sondern lediglich 8 int-Werte umfassen würde. Über `set_next_size()` kann direkt Einfluss auf die Größe der Speicherblocks genommen werden. Wenn Sie lediglich eine maximale Größe vorgeben möchten, übergeben Sie diese und nicht 0 als zweiten Parameter an den Konstruktor.
Mit `boost::singleton_pool` bietet Boost.Pool eine Klasse an, die zwischen `boost::simple_segregated_storage` und `boost::object_pool` angesiedelt ist. Sehen Sie sich dazu Beispiel 4.4 an.

Beispiel 4.4 Segmentierter Speicher als Singleton mit `boost::singleton_pool`

```
#include <boost/pool/singleton_pool.hpp>

struct int_pool {};
typedef boost::singleton_pool<int_pool, sizeof(int)> singleton_int_pool;

int main()
{
  int *i = static_cast<int*>(singleton_int_pool::malloc());
  *i = 1;

  int *j = static_cast<int*>(singleton_int_pool::ordered_malloc(10));
  j[9] = 2;

  singleton_int_pool::release_memory();
  singleton_int_pool::purge_memory();
}
```

`boost::singleton_pool` ist in der Headerdatei `boost/pool/singleton_pool.hpp` definiert. Die Klasse ähnelt `boost::simple_segregated_storage`, weil die Segmentgröße als Template-Parameter angegeben wird und nicht der Typ der Objekte, der gespeichert werden soll. Daher geben Methoden wie `malloc()` und `ordered_malloc()` einen Zeiger vom Typ void* zurück, der explizit gecastet werden muss.
Die Klasse ähnelt jedoch auch `boost::object_pool`, da sie automatisch Speicher anfordert. Die Größe des nächsten Speicherblocks und eine eventuelle maximale Größe müssen jedoch als Template-Parameter angege-

ben werden. In dieser Hinsicht unterscheidet sich `boost::singleton_pool` von `boost::object_pool`: Sie können bei `boost::singleton_pool` nicht zur Laufzeit die Größe des nächsten Speicherblocks vorgeben oder ändern.
Sie können `boost::singleton_pool` mehrfach verwenden, um mehrere segmentierte Speicher zu verwalten. Als ersten Template-Parameter übergeben Sie an `boost::singleton_pool` einen *Tag*. Es handelt sich dabei um einen beliebigen Typ, der als Name für den entsprechenden segmentierten Speicher dient. Im Beispiel 4.4 wird die Struktur `int_pool` als Tag verwendet, um anzudeuten, dass `singleton_int_pool` Speicher für ints verwaltet. Dank dieser Tags können Sie sicherstellen, dass mehrere Singletons unterschiedliche Speicher verwalten, auch wenn der zweite Template-Parameter, der die Segmentgröße beschreibt, jeweils gleich ist. Der Tag hat keine andere Bedeutung als unterschiedliche Instanzen von `boost::singleton_pool` zu erstellen, die jeweils ihren eigenen Speicher verwalten.
Zur Speicherfreigabe bietet `boost::singleton_pool` zwei Methoden an: Mit `release_memory()` geben Sie alle Speicherblöcke frei, die im Moment nicht verwendet werden. Mit `purge_memory()` geben Sie alle Speicherblöcke frei – auch die, die im Moment verwendet werden. Der Aufruf von `purge_memory()` setzt einen Singleton-Pool in den Ausgangszustand zurück.
`release_memory()` und `purge_memory()` geben Speicher ans Betriebssystem zurück. Freigabe bedeutet hier nicht, dass Segmente wieder zur Verfügung stehen, weil `boost::singleton_pool` den intern verwalteten Speicher als verfügbar markiert. Um lediglich reservierte Segmente an `boost::singleton_pool` zurückzugeben, müssen Methoden wie `free()` oder `ordered_free()` aufgerufen werden.
`boost::object_pool` und `boost::singleton_pool` ermöglichen Ihnen, Speicher explizit anzufragen, wenn Sie Speicher benötigen. So müssen Sie Methoden wie `malloc()` oder `construct()` aufrufen. Mit `boost::pool_allocator` bietet Boost.Pool jedoch auch eine Klasse an, die Sie als Allokator an Container übergeben können. Sehen Sie sich dazu Beispiel 4.5 an.

Beispiel 4.5 Segmentierter Speicher mit `boost::pool_allocator`

```
#include <boost/pool/pool_alloc.hpp>
#include <vector>

int main()
{
  std::vector<int, boost::pool_allocator<int>> v;
  for (int i = 0; i < 1000; ++i)
    v.push_back(i);

  v.clear();
  boost::singleton_pool<boost::pool_allocator_tag, sizeof(int)>::
    purge_memory();
}
```

`boost::pool_allocator` ist in der Headerdatei `boost/pool/pool_alloc.hpp` definiert. Es handelt sich um einen Allokator, der Containern aus der Standardbibliothek üblicherweise als zweiter Template-Parameter übergeben werden kann. Der Allokator stellt dem Container den benötigten Speicher zur Verfügung.
`boost::pool_allocator` basiert intern auf `boost::singleton_pool`. Um beispielsweise Speicher freizugeben, müssen Sie wie im Beispiel 4.4 mit einem Tag auf `boost::singleton_pool` zugreifen und `purge_memory()` oder `release_memory()` aufrufen. Im Beispiel 4.5 wird als Tag `boost::pool_allocator_tag` verwendet. Es handelt sich dabei um einen Tag, der von Boost.Pool definiert ist und von `boost::pool_allocator` für den intern verwendeten `boost::singleton_pool` benutzt wird.
Wenn im Beispiel 4.5 zum ersten Mal `push_back()` aufgerufen wird, greift **v** auf den Allokator zu, um den notwendigen Speicher zu erhalten. Da als Allokator `boost::pool_allocator` zum Einsatz kommt, wird ein Speicherblock in einer Größe von 32 int-Werten reserviert und **v** ein Zeiger auf das erste Segment in diesem Speicherblock übergeben. Das Segment hat die Größe von einem int. Jeder weitere Aufruf von `push_back()` wird aus diesem ersten Speicherblock bedient, bis der Allokator feststellt, dass er einen größeren Speicherblock benötigt.
Beachten Sie, dass Sie einen Container mit `clear()` löschen sollten, bevor Sie Speicher wie im Beispiel 4.5 mit `purge_memory()` freigeben. Ein Aufruf von `purge_memory()` gibt zwar Speicher frei, teilt dem entsprechenden Container aber nicht mit, dass ihm der Speicher nicht mehr gehört. Ein Aufruf von `release_memory()` ist weniger gefährlich, da nur Speicherblöcke freigegeben werden, die im Moment nicht benötigt werden.
Neben `boost::pool_allocator` bietet Boost.Pool auch einen Allokator namens `boost::fast_pool_allocator` an. Sehen Sie sich dazu Beispiel 4.6 an.

Beispiel 4.6 Segmentierter Speicher mit `boost::fast_pool_allocator`

```
#define BOOST_POOL_NO_MT
#include <boost/pool/pool_alloc.hpp>
#include <list>

int main()
{
  typedef boost::fast_pool_allocator<int,
    boost::default_user_allocator_new_delete,
    boost::details::pool::default_mutex,
    64, 128> allocator;

  std::list<int, allocator> l;
  for (int i = 0; i < 1000; ++i)
    l.push_back(i);

  l.clear();
  boost::singleton_pool<boost::fast_pool_allocator_tag, sizeof(int)>::
    purge_memory();
}
```

Sie verwenden beide Allokatoren grundsätzlich auf die gleiche Art und Weise. `boost::pool_allocator` sollten Sie vorziehen, wenn mehrere aufeinander folgende Segmente auf einmal angefordert werden müssen. `boost::fast_pool_allocator` können Sie einsetzen, wenn Segmente jeweils einzeln eines nach dem anderen angefordert werden. Grob vereinfacht: Für `std::vector` verwenden Sie `boost::pool_allocator`, für `std::list` `boost::fast_pool_allocator`.
Beispiel 4.6 zeigt Ihnen, welche Template-Parameter Sie an `boost::fast_pool_allocator` übergeben können. `boost::pool_allocator` akzeptiert die gleichen Parameter.
Bei `boost::default_user_allocator_new_delete` handelt es sich um eine Klasse, die mit `new` Speicherblöcke reserviert und mit `delete[]` freigibt. Sie können alternativ `boost::default_user_allocator_malloc_free` verwenden. Diese Klasse ruft `malloc()` und `free()` auf.
boost::details::pool::default_mutex ist eine Typdefinition, die entweder auf `boost::mutex` oder auf `boost::details::pool::null_mutex` gesetzt ist. Standardmäßig wird `boost::mutex` verwendet, damit mehrere Threads auf einen segmentierten Speicher zugreifen können. Wenn wie im Beispiel 4.6 das Makro `BOOST_POOL_NO_MT` definiert ist, wird die Multithreading-Unterstützung für Boost.Pool explizit deaktiviert. Im Beispiel 4.6 wird demnach vom Allokator ein Null-Mutex verwendet.
Die letzten beiden Parameter, die im Beispiel 4.6 an `boost::fast_pool_allocator` übergeben werden, legen die Größe des ersten Speicherblocks sowie die maximale Größe eines Speicherblocks fest.

Teil II

Stringverarbeitung

Die folgenden Boost-Bibliotheken stellen viele Hilfsmittel zur einfacheren Verarbeitung von Strings zur Verfügung.

- Boost.StringAlgorithms bietet Algorithmen speziell für Strings an. So existieren zum Beispiel Algorithmen, um Strings in Groß- oder Kleinbuchstaben umzuwandeln.
- Boost.LexicalCast bietet einen Cast-Operator an, um eine Zahl in einen String oder in die andere Richtung umzuwandeln. Die Bibliothek verwendet intern String-Streams, kann jedoch für Umwandlungen zwischen bestimmten Typen optimiert sein.
- Boost.Format bietet einen typsicheren Ersatz für `std::printf()`, um Daten formatiert als String auszugeben. Die Bibliothek verwendet so wie Boost.LexicalCast intern String-Streams. Boost.Format ist erweiterbar und unterstützt benutzerdefinierte Typen, wenn für diese Stream-Operatoren definiert sind.
- Boost.Regex und Boost.Xpressive sind Bibliotheken, um Strings mit regulären Ausdrücken durchsuchen zu können. Während Boost.Regex erwartet, dass reguläre Ausdrücke als String angegeben werden, können sie mit Boost.Xpressive als C++-Code geschrieben werden.
- Boost.Tokenizer ermöglicht, über Substrings in einem String zu iterieren.
- Boost.Spirit kann verwendet werden, um Parser zu entwickeln, die auf Regeln ähnlich der Erweiterten Backus-Naur-Form basieren.

Kapitel 5

Boost.StringAlgorithms

Die Bibliothek Boost.StringAlgorithms stellt viele freistehende Funktionen im Namensraum `boost::algorithm` zur Verfügung, mit denen sich Strings verarbeiten lassen. Strings können dabei vom Typ `std::string`, `std::wstring` oder einer anderen Instanz der Template-Klasse `std::basic_string` sein. Dazu zählen auch die mit C++11 eingeführten Klassen `std::u16string` und `std::u32string`.
Alle Funktionen sind je nach Kategorie in unterschiedlichen Headerdateien definiert. So muss, um auf Funktionen zum Umwandeln von Groß- und Kleinbuchstaben zuzugreifen, die Headerdatei `boost/algorithm/string/case_conv.hpp` eingebunden werden. Weil Boost.StringAlgorithms mehr als 20 Kategorien und somit ebenso viele Headerdateien anbietet, steht mit `boost/algorithm/string.hpp` eine Master-Headerdatei zur Verfügung, die alle anderen einbindet.

Beispiel 5.1 String in Großbuchstaben umwandeln

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout << to_upper_copy(s) << '\n';
}
```

Die Funktion `boost::algorithm::to_upper_copy()` kann verwendet werden, um einen String in Großbuchstaben zu konvertieren. Neben dieser Funktion stellt Boost.StringAlgorithms auch `boost::algorithm::to_lower_copy()` zur Verfügung, um einen String in Kleinbuchstaben umzuwandeln. Beide Funktionen geben den umgewandelten String als Ergebnis zurück. Soll der String selbst, der als Parameter an die Funktion übergeben wird, umgewandelt werden, müssen `boost::algorithm::to_upper()` oder `boost::algorithm::to_lower()` aufgerufen werden.
Im Beispiel 5.1 wird der String „Boost C++ Libraries“ mit `boost::algorithm::to_upper_copy()` in Großbuchstaben umgewandelt. Wenn Sie das Beispiel ausführen, wird `BOOST C++ LIBRARIES` ausgegeben.
Die von Boost.StringAlgorithms angebotenen Funktionen berücksichtigen Locales. Funktionen wie `boost::algorithm::to_upper_copy()` verwenden das globale Locale, um einen String in Großbuchstaben umzuwandeln, wenn ein Locale nicht explizit als Parameter übergeben wird.

Beispiel 5.2 String mit einem Locale in Großbuchstaben umwandeln

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <locale>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ k\xfct\xfcphaneleri";
```

```
  std::string upper_case1 = to_upper_copy(s);
  std::string upper_case2 = to_upper_copy(s, std::locale{"Turkish"});
  std::locale::global(std::locale{"Turkish"});
  std::cout << upper_case1 << '\n';
  std::cout << upper_case2 << '\n';
}
```

Im Beispiel 5.2 wird `boost::algorithm::to_upper_copy()` zweimal aufgerufen, um den String „Boost C++ kütüphaneleri“ in Türkisch in Großbuchstaben umzuwandeln. Der erste Aufruf von `boost::algorithm::to_upper_copy()` verwendet das globale Locale. Da das globable Locale im Beispiel vor dem ersten Aufruf von `boost::algorithm::to_upper_copy()` nicht geändert wurde, ist dies das C-Locale. Da der zu umwandelnde String Umlaute enthält, die für das C-Locale keine Buchstaben sind, wird als Ergebnis für den ersten Aufruf von `boost::algorithm::to_upper_copy()` `BOOST C++ KüTüPHANELERI` ausgegeben.
Der zweite Aufruf findet mit einem Locale für den türkischen Kulturkreis statt. Die Umlaute werden in diesem Fall in Großbuchstaben umgewandelt. Als Ergebnis wird für den zweiten Aufruf von `boost::algorithm::to_upper_copy()` `BOOST C++ KÜTÜPHANELERI` ausgegeben.

> **Anmerkung**
>
> Ersetzen Sie die Angabe „Turkish“ mit „tr_TR“, wenn Sie das Beispiel auf einem POSIX-Betriebssystem ausführen möchten. Stellen Sie außerdem sicher, dass das Locale für den türkischen Sprach- und Kulturkreis installiert ist.

Beispiel 5.3 String-Algorithmen, um Zeichen in einem String zu löschen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout << erase_first_copy(s, "s") << '\n';
  std::cout << erase_nth_copy(s, "s", 0) << '\n';
  std::cout << erase_last_copy(s, "s") << '\n';
  std::cout << erase_all_copy(s, "s") << '\n';
  std::cout << erase_head_copy(s, 5) << '\n';
  std::cout << erase_tail_copy(s, 9) << '\n';
}
```

Boost.StringAlgorithms stellt wie im Beispiel 5.3 zu sehen mehrere Funktionen zur Verfügung, um Zeichen in einem String zu löschen. Dabei kann auf unterschiedliche Weise angegeben werden, welcher Teil eines Strings gelöscht werden soll. So kann zum Beispiel mit `boost::algorithm::erase_all_copy()` der Buchstabe „s“ aus einem String komplett oder mit `boost::algorithm::erase_first_copy()` nur das erste „s“ gelöscht werden. Außerdem stehen mit `boost::algorithm::erase_head_copy()` und `boost::algorithm::erase_tail_copy()` Funktionen zur Verfügung, die den Anfang oder das Ende eines Strings um eine bestimmte Anzahl an Zeichen kürzen.

Beispiel 5.4 Mit `boost::algorithm::find_first()` nach Substrings suchen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
```

```
  boost::iterator_range<std::string::iterator> r = find_first(s, "C++");
  std::cout << r << '\n';
  r = find_first(s, "xyz");
  std::cout << r << '\n';
}
```

Verschiedene Funktionen wie `boost::algorithm::find_first()` stehen zur Verfügung, um Strings nach einem Substring zu durchsuchen. So kann auch mit `boost::algorithm::find_last()`, `boost::algorithm::find_nth()`, `boost::algorithm::find_head()` und `boost::algorithm::find_tail()` nach Substrings gesucht werden.
Allen genannten Funktionen ist gemein, dass sie ein paar Paar Iteratoren zurückgeben. Dieses Paar hat den Typ `boost::iterator_range`. Diese Klasse stammt aus der Bibliothek Boost.Range, die basierend auf dem Iterator-Konzept ein Range-Konzept definiert. Da der Operator `operator<<` für `boost::iterator_range` überladen ist, kann das Ergebnis der Suchalgorithmen direkt auf die Standardausgabe ausgegeben werden. So wird im Beispiel 5.4 als erstes Ergebnis `C++` und als zweites Ergebnis ein leerer String ausgegeben.

Beispiel 5.5 Strings mit `boost::algorithm::join()` verketten

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<std::string> v{"Boost", "C++", "Libraries"};
  std::cout << join(v, " ") << '\n';
}
```

Der Funktion `boost::algorithm::join()` wird als erster Parameter ein Container mit Strings übergeben. Die Funktion hängt die Strings im Container aneinander und fügt zwischen zwei Strings jeweils den String ein, der als zweiter Parameter übergeben wird. So gibt Beispiel 5.5 `Boost C++ Libraries` aus.

Beispiel 5.6 String-Algorithmen, um Zeichen in einem String zu ersetzen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout << replace_first_copy(s, "+", "-") << '\n';
  std::cout << replace_nth_copy(s, "+", 0, "-") << '\n';
  std::cout << replace_last_copy(s, "+", "-") << '\n';
  std::cout << replace_all_copy(s, "+", "-") << '\n';
  std::cout << replace_head_copy(s, 5, "BOOST") << '\n';
  std::cout << replace_tail_copy(s, 9, "LIBRARIES") << '\n';
}
```

So wie Boost.StringAlgorithms mehrere Funktionen zur Verfügung stellt, um Strings zu durchsuchen oder eine Zeichenkette in Strings zu löschen, stehen auch Funktionen zur Verfügung, um eine Zeichenkette in einem String mit einer anderen Zeichenkette zu ersetzen. Zu diesen Funktionen zählen `boost::algorithm::replace_first_copy()`, `boost::algorithm::replace_nth_copy()`, `boost::algorithm::replace_last_copy()`, `boost::algorithm::replace_all_copy()`, `boost::algorithm::replace_head_copy()` und `boost::algorithm::replace_tail_copy()`. Wie im Beispiel 5.6 zu sehen, werden diese Funktionen genauso angewandt wie die Funktionen zum Suchen und Löschen von Zeichenketten. Der einzige Unterschied ist, dass sie einen zusätzlichen Parameter erwarten – die Zeichenkette, die die gesuchte ersetzen soll.

Beispiel 5.7 String-Algorithmen, um Strings zu trimmen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "\t Boost C++ Libraries \t";
  std::cout << "_" << trim_left_copy(s) << "_\n";
  std::cout << "_" << trim_right_copy(s) << "_\n";
  std::cout << "_" << trim_copy(s) << "_\n";
}
```

Wenn eine Zeichenkette an Anfang und Ende automatisch um Leerzeichen gekürzt werden soll, können wie im Beispiel 5.7 die Funktionen `boost::algorithm::trim_left_copy()`, `boost::algorithm::trim_right_copy()` und `boost::algorithm::trim_copy()` verwendet werden. Welche Zeichen als Leerzeichen gelten, hängt vom globalen Locale ab.
Für verschiedene Funktionen ermöglicht Boost.StringAlgorithms, als zusätzlichen Parameter ein Prädikat zu übergeben, von dem abhängt, auf welche Zeichen in einem String die Funktion angewandt wird. So stehen neben den eben genannten drei Funktionen `boost::algorithm::trim_left_copy_if()`, `boost::algorithm::trim_right_copy_if()` und `boost::algorithm::trim_copy_if()` zur Verfügung.

Beispiel 5.8 Prädikate mit `boost::algorithm::is_any_of()` erstellen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "--Boost C++ Libraries--";
  std::cout << trim_left_copy_if(s, is_any_of("-")) << '\n';
  std::cout << trim_right_copy_if(s, is_any_of("-")) << '\n';
  std::cout << trim_copy_if(s, is_any_of("-")) << '\n';
}
```

Im Beispiel 5.8 wird zusätzlich auf die Funktion `boost::algorithm::is_any_of()` zugegriffen. Es handelt sich hierbei um eine Hilfsfunktion, mit der ein Prädikat erstellt wird, das überprüft, ob ein Zeichen in dem String vorkommt, der als Parameter an `boost::algorithm::is_any_of()` übergeben wird. Mit Hilfe von `boost::algorithm::is_any_of()` kann angegeben werden, um welche Zeichen ein String getrimmt werden soll. Im Beispiel 5.8 ist das der Bindestrich.
Boost.StringAlgorithms stellt zahlreiche Hilfsfunktionen zur Verfügung, die häufig benötigte Prädikate zurückgeben.

Beispiel 5.9 Prädikate mit `boost::algorithm::is_digit()` erstellen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "123456789Boost C++ Libraries123456789";
  std::cout << trim_left_copy_if(s, is_digit()) << '\n';
  std::cout << trim_right_copy_if(s, is_digit()) << '\n';
  std::cout << trim_copy_if(s, is_digit()) << '\n';
}
```

Das Prädikat, das von `boost::algorithm::is_digit()` zurückgegeben wird, gibt für alle Ziffern `true` zurück. So werden im Beispiel 5.9 Ziffern aus dem String **s** entfernt.
Neben `boost::algorithm::is_digit()` stehen mit `boost::algorithm::is_upper()` und `boost::algorithm::is_lower()` Hilfsfunktionen zur Verfügung, die Zeichen auf Groß- und Kleinschreibung überprüfen. All diese Funktionen verwenden das globale Locale, wenn kein Locale explizit als Parameter übergeben wird.

Beispiel 5.10 String-Algorithmen, um Strings mit anderen zu vergleichen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout.setf(std::ios::boolalpha);
  std::cout << starts_with(s, "Boost") << '\n';
  std::cout << ends_with(s, "Libraries") << '\n';
  std::cout << contains(s, "C++") << '\n';
  std::cout << lexicographical_compare(s, "Boost") << '\n';
}
```

Neben den Prädikaten, die mit Hilfe der eben vorgestellten Hilfsfunktionen erstellt werden und die einzelne Zeichen überprüfen, bietet Boost.StringAlgorithms Funktionen an, mit denen Strings überprüft werden können. Die Funktionen `boost::algorithm::starts_with()`, `boost::algorithm::ends_with()`, `boost::algorithm::contains()` und `boost::algorithm::lexicographical_compare()` können wie im Beispiel 5.10 aufgerufen werden, um zwei Strings miteinander zu vergleichen.

Beispiel 5.11 Strings mit `boost::algorithm::split()` teilen

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::vector<std::string> v;
  split(v, s, is_space());
  std::cout << v.size() << '\n';
}
```

Die Funktion `boost::algorithm::split()` wird verwendet, um einen String an bestimmten Stellen in seine Teile zu zerlegen und diese in einem Container zu speichern. Der Funktion muss als dritter Parameter ein Prädikat übergeben werden, das für jedes Zeichen entscheidet, ob der String an dieser Stelle geteilt werden soll oder nicht. Hier können die von Boost.StringAlgorithms zur Verfügung gestellten Hilfsfunktionen verwendet werden, die Prädikate zurückgeben. Im Beispiel 5.11 wird auf `boost::algorithm::is_space()` zugegriffen, so dass der String an Leerzeichen gesplittet wird.
Viele der Funktionen, die Sie in diesem Kapitel kennengelernt haben, existieren auch in einer Variante, die Groß- und Kleinschreibung ignoriert. Diese Funktionen haben typischerweise den gleichen Namen, beginnen aber mit einem „i“. So heißt die zu `boost::algorithm::erase_all_copy()` entsprechende Funktion, die Zeichenketten unabhängig von Groß- und Kleinschreibung löscht, `boost::algorithm::ierase_all_copy()`.
Abschließend soll erwähnt werden, dass Boost.StringAlgorithms für mehrere Funktionen auch reguläre Ausdrücke unterstützt. So wird im Beispiel 5.12 mit `boost::algorithm::find_regex()` nach einem regulären Ausdruck gesucht.

Beispiel 5.12 Strings mit `boost::algorithm::find_regex()` durchsuchen

```
#include <boost/algorithm/string.hpp>
#include <boost/algorithm/string/regex.hpp>
#include <string>
```

```
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  boost::iterator_range<std::string::iterator> r =
    find_regex(s, boost::regex{"\\w\\+\\+"});
  std::cout << r << '\n';
}
```

Für den regulären Ausdruck wird auf eine Klasse `boost::regex` zugegriffen, die in der Bibliothek Boost.Regex definiert ist. Boost.Regex wird im Kapitel 8 vorgestellt.
Wenn Sie Beispiel 5.12 ausführen, wird `C++` ausgegeben.

Kapitel 6

Boost.LexicalCast

Boost.LexicalCast bietet einen Cast-Operator `boost::lexical_cast` an, mit dem Zahlen von Strings in numerische Typen wie int oder double umgewandelt werden können – und umgekehrt. `boost::lexical_cast` stellt somit eine Alternative zu den seit C++11 in der Standardbibliothek verfügbaren Funktionen wie `std::stoi()`, `std::stod()` oder `std::to_string()` dar.

Beispiel 6.1 `boost::lexical_cast` in Aktion

```
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = boost::lexical_cast<std::string>(123);
  std::cout << s << '\n';
  double d = boost::lexical_cast<double>(s);
  std::cout << d << '\n';
}
```

Beispiel 6.1 wandelt die Ganzzahl 123 in einen String um, um sie anschließend vom String in eine Kommazahl zu konvertieren. Um wie im Beispiel den Cast-Operator `boost::lexical_cast` einsetzen zu können, muss die Headerdatei `boost/lexical_cast.hpp` eingebunden werden.
`boost::lexical_cast` verwendet intern Streams, um Umwandlungen durchzuführen. Deswegen können grundsätzlich nur Typen umgewandelt werden, für die die Operatoren `operator<<` und `operator>>` überladen sind. `boost::lexical_cast` kann jedoch für einige Typen spezialisiert sein und effizientere Typumwandlungen anbieten.
Schlägt eine Umwandlung fehl, wird eine Ausnahme vom Typ `boost::bad_lexical_cast` geworfen. Diese Klasse ist von `std::bad_cast` abgeleitet.

Beispiel 6.2 `boost::bad_lexical_cast` im Fehlerfall

```
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

int main()
{
  try
  {
    int i = boost::lexical_cast<int>("abc");
    std::cout << i << '\n';
  }
  catch (const boost::bad_lexical_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Beispiel 6.2 wirft eine Ausnahme, weil der String „abc“ nicht in eine Zahl vom Typ int umgewandelt werden kann.

Kapitel 7

Boost.Format

Boost.Format bietet einen Ersatz für die Funktion `std::printf()` an. Diese Funktion stammt aus dem C-Standard und ermöglicht eine formatierte Datenausgabe. Sie ist weder typsicher noch erweiterbar. Mit Boost.Format steht eine typsichere und erweiterbare Alternative zur Verfügung.
Boost.Format bietet eine Klasse `boost::format` an, die in der Headerdatei `boost/format.hpp` definiert ist. Dem Konstruktor dieser Klasse wird ähnlich wie `std::printf()` ein String übergeben, der Sonderzeichen zur Formatierung enthält. Werte, die die Sonderzeichen ersetzen sollen, werden mit dem Operator `operator%` verknüpft.

Beispiel 7.1 Formatierte Datenausgabe mit `boost::format`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%1%.%2%.%3%"} % 12 % 5 % 2014 << '\n';
}
```

Als Platzhalter werden von Boost.Format Zahlen verwendet, die zwischen zwei Prozentzeichen stehen. Mit diesen Zahlen wird auf Werte verwiesen, die mit dem Operator `operator%` verknüpft werden. So werden im Beispiel 7.1 die Zahlen 12, 5 und 2014 zum Datum `12.5.2014` verknüpft. Wenn wie in den USA der Monat vor dem Tag stehen soll, können anstatt den Zahlen die Platzhalter ausgetauscht werden.

Beispiel 7.2 Nummerierte Platzhalter mit `boost::format`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%2%/%1%/%3%"} % 12 % 5 % 2014 << '\n';
}
```

Beispiel 7.2 gibt als Ergebnis `5/12/2014` aus.
Um Daten zu formatieren, können mit Hilfe der Funktion `boost::io::group()`, die ebenfalls von Boost.Format zur Verfügung gestellt wird, die aus dem Standard bekannten Manipulatoren angewandt werden.

Beispiel 7.3 Manipulatoren mit `boost::io::group()` verwenden

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%1% %2% %1%"} %
    boost::io::group(std::showpos, 1) % 2 << '\n';
}
```

Beispiel 7.3 gibt als Ergebnis `+1 2 +1` aus. Weil der Manipulator `std::showpos()` mit `boost::io::group()` auf die Zahl 1 angewandt wird, wird das Pluszeichen immer dann mitangegeben, wenn 1 ausgegeben werden soll.
Soll nur die erste Ausgabe von 1 ein Pluszeichen mitanzeigen, muss das Sonderzeichen zur Formatierung angepasst werden.

Beispiel 7.4 Platzhalter mit Sonderzeichen

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%|1$+| %2% %1%"} % 1 % 2 << '\n';
}
```

Der Platzhalter „%1%“ wird im Beispiel 7.4 durch „%|1$+|“ ersetzt. Wenn die Formatierung angepasst werden soll, erfordert dies nicht nur zwei zusätzliche vertikale Balken. Der Verweis auf eine Variable muss ebenfalls zwischen die vertikalen Balken gesetzt werden und erfolgt nicht mehr mit „1%“, sondern mit „1$“. Die Änderungen sind notwendig, um das Pluszeichen unterzubringen, damit als Ergebnis `+1 2 1` erhalten wird.
Beachten Sie, dass Verweise auf Variablen optional sind. Sie müssen sich jedoch für alle Platzhalter entscheiden, ob Sie Verweise angeben wollen oder nicht. So ist im Beispiel 7.5 für einen von drei Platzhaltern kein Verweis angegeben, was bei der Ausführung zu einem Fehler führt.

Beispiel 7.5 `boost::io::format_error` im Fehlerfall

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::format{"%|+| %2% %1%"} % 1 % 2 << '\n';
  }
  catch (boost::io::format_error &ex)
  {
    std::cout << ex.what() << '\n';
  }
}
```

Im Beispiel 7.5 wird eine Ausnahme vom Typ `boost::io::format_error` geworfen. Genaugenommen hat die Ausnahme den Typ `boost::io::bad_format_string`. Da die verschiedenen Ausnahmen von `boost::io::format_error` abgeleitet sind, ist es einfacher, Ausnahmen von diesem Typ abzufangen.
Beispiel 7.6 zeigt, wie das Programm ohne Verweise auf Variablen geschrieben werden muss.

Beispiel 7.6 Platzhalter ohne Nummerierung

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%|+| %|| %||"} % 1 % 2 % 1 << '\n';
}
```

Wenn Sie auf die Angabe der vertikalen Balken, die in diesem Fall für den zweiten und dritten Platzhalter die Formatierung nicht näher spezifizieren, verzichten möchten, können Sie dies tun. Sie erhalten dann eine Syntax, die der von `std::printf()` sehr ähnlich ist.

Beispiel 7.7 `boost::format` mit der von `std::printf()` gewohnten Syntax

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
```

```
  std::cout << boost::format{"%+d %d %d"} % 1 % 2 % 1 << '\n';
}
```

Beachten Sie, dass die Formatierung der von `std::printf()` ähnlich sieht, Boost.Format jedoch den Vorteil der Typsicherheit bietet. So bedeutet der Buchstabe „d“ im Beispiel 7.7 nicht, dass eine Dezimalzahl ausgegeben werden muss, sondern dass der Manipulator `std::dec()` auf den Stream angewandt wird, den `boost::format` intern verwendet. Auf diese Weise können Formatierungen angegeben werden, die für `std::printf()` ungültig wären und zu einem Laufzeitfehler führen würden.

Beispiel 7.8 `boost::format` mit vermeintlich ungültigen Platzhaltern

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%+s %s %s"} % 1 % 2 % 1 << '\n';
}
```

Während für `std::printf()` der Buchstabe „s“ lediglich für Strings verwendet werden darf – also den Typ const char* in C – funktioniert Beispiel 7.8 einwandfrei. Boost.Format erwartet nicht zwingend Strings, sondern setzt lediglich entsprechende Manipulatoren ein, um den internen Stream zu konfigurieren.
Boost.Format ist nicht nur typsicher, sondern auch erweiterbar. So können Sie Objekte beliebiger Typen mit Boost.Format verwenden, solange der Operator `operator<<` für `std::ostream` überladen ist.

Beispiel 7.9 `boost::format` mit benutzerdefiniertem Typ

```
#include <boost/format.hpp>
#include <string>
#include <iostream>

struct animal
{
  std::string name;
  int legs;
};

std::ostream &operator<<(std::ostream &os, const animal &a)
{
  return os << a.name << ',' << a.legs;
}

int main()
{
  animal a{"cat", 4};
  std::cout << boost::format{"%1%"} % a << '\n';
}
```

Beispiel 7.9 gibt ein Objekt von einem benutzerdefinierten Typ `animal` über `boost::format` auf die Standardausgabe aus. Dies ist möglich, weil der Stream-Operator für `animal` überladen ist.

Kapitel 8

Boost.Regex

Boost.Regex ermöglicht es, *reguläre Ausdrücke* in C++ einzusetzen. Da die Bibliothek seit C++11 Teil der Standardbibliothek ist, sind Sie nicht auf Boost.Regex angewiesen, wenn Sie in einer Entwicklungsumgebung arbeiten, die C++11 unterstützt. Sie können auf die gleichnamigen Klassen und Funktionen im Namensraum `std` zugreifen, wenn Sie die Headerdatei `regex` einbinden.
Die wichtigsten Klassen in Boost.Regex sind `boost::regex` und `boost::smatch`, die beide in der Headerdatei `boost/regex.hpp` definiert sind. Während `boost::regex` verwendet wird, um einen regulären Ausdruck zu definieren, speichert `boost::smatch` Suchergebnisse.
Im Folgenden lernen Sie die drei Funktionen kennen, die Boost.Regex zur Suche mit regulären Ausdrücken anbietet.

Beispiel 8.1 Strings mit `boost::regex_match()` vergleichen

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"\\w+\\s\\w+"};
  std::cout << std::boolalpha << boost::regex_match(s, expr) << '\n';
}
```

Die Funktion `boost::regex_match()`, die im Beispiel 8.1 verwendet wird, vergleicht einen String in Gänze mit einem regulären Ausdruck. Die Funktion gibt `true` zurück, wenn der reguläre Ausdruck auf den gesamten String passt.
Um einen String nach einem regulären Ausdruck zu durchsuchen, verwenden Sie `boost::regex_search()`.

Beispiel 8.2 Strings mit `boost::regex_search()` durchsuchen

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  boost::smatch what;
  if (boost::regex_search(s, what, expr))
  {
    std::cout << what[0] << '\n';
    std::cout << what[1] << "_" << what[2] << '\n';
  }
}
```

Die Funktion `boost::regex_search()` erwartet als zusätzlichen Parameter eine Referenz auf ein Objekt vom Typ `boost::smatch`. In diesem Objekt speichert `boost::regex_search()` die Ergebnisse. Dabei wird nur

nach Gruppierungen gesucht. So werden im Beispiel 8.2 zwei Ergebnisse gespeichert, weil es zwei Gruppierungen im regulären Ausdruck gibt.
Die Klasse `boost::smatch`, in der Suchergebnisse gespeichert werden, ist ein Container für Elemente vom Typ `boost::sub_match`. Sie bietet eine ähnliche Schnittstelle wie `std::vector`. So kann über `operator[]` auf Elemente vom Typ `boost::sub_match` zugegriffen werden.
`boost::sub_match` speichert Iteratoren auf die Positionen in einem String, die einer Gruppierung in einem regulären Ausdruck entsprechen. Da `boost::sub_match` von `std::pair` abgeleitet ist, kann über **`first`** und **`second`** auf die Iteratoren zugegriffen werden, die einen Substring identifizieren. Wie im Beispiel 8.2 zu sehen, muss nicht auf diese Iteratoren zugegriffen werden, um einen Substring auf die Standardausgabe auszugeben. Da der Operator `operator<<` für `boost::sub_match` überladen ist, kann ein Substring direkt ausgegeben werden.
Beachten Sie, dass `boost::sub_match` Iteratoren speichert. Ergebnisse werden nicht kopiert. Sie dürfen daher auf Iteratoren in Objekten vom Typ `boost::sub_match` nur zugreifen, solange der String, auf den die Iteratoren zeigen, existiert.
Beachten Sie außerdem, dass das erste Element im Container `boost::smatch` mit dem Index 0 Iteratoren speichert, die auf den String zeigen, der dem gesamten regulären Ausdruck entspricht. Der erste Substring, der der ersten Gruppierung entspricht, wird über den Index 1 angesprochen.
Die dritte Funktion, die Boost.Regex bietet, ist `boost::regex_replace()`.

Beispiel 8.3 Zeichen in Strings mit `boost::regex_replace()` ersetzen

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = " Boost Libraries ";
  boost::regex expr{"\\s"};
  std::string fmt{"_"};
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

Der Funktion `boost::regex_replace()` muss neben dem zu durchsuchenden String und einem regulären Ausdruck ein Formatierungsstring übergeben werden. Der Formatierungsstring definiert, wie Substrings, die Gruppierungen im regulären Ausdruck entsprechen, ersetzt werden. Wenn der reguläre Ausdruck so wie im Beispiel 8.3 keine Gruppierungen enthält, werden entsprechende Substrings eins zu eins mit dem Formatierungsstring ersetzt. Beispiel 8.3 gibt als Ergebnis `_Boost_Libraries_` aus.
Beachten Sie, dass `boost::regex_replace()` den gesamten String nach einem regulären Ausdruck durchsucht. So werden im Beispiel 8.3 alle drei Leerzeichen durch einen Unterstrich ersetzt.

Beispiel 8.4 Formatierungsstring mit Referenzen auf Gruppierungen

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  std::string fmt{"\\2 \\1"};
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

Im Formatierungsstring kann auf Substrings zugegriffen werden, die durch Gruppierungen im regulären Ausdruck zurückgegeben werden. So können wie im Beispiel 8.4 die Positionen der beiden Wörter getauscht werden, so dass als Ergebnis `Libraries Boost` ausgegeben wird.
Beachten Sie, dass es verschiedene Standards für reguläre Ausdrücke und Formatierungsstrings gibt. Für alle drei vorgestellten Funktionen kann ein weiterer Parameter übergeben werden, mit dem ein bestimmter Standard ausgewählt werden kann. Außerdem kann zum Beispiel angegeben werden, dass Sonderzeichen in einem Formatierungsstring nicht interpretiert werden sollen, sondern der Formatierungsstring komplett den String ersetzen soll, auf den der reguläre Ausdruck zutrifft.

Beispiel 8.5 Flags für Formatierungsstrings angeben

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  std::string fmt{"\\2 \\1"};
  std::cout << boost::regex_replace(s, expr, fmt,
    boost::regex_constants::format_literal) << '\n';
}
```

Im Beispiel 8.5 wird das Flag `boost::regex_constants::format_literal` als vierter Parameter an `boost::regex_replace()` übergeben, um die Interpretation der Sonderzeichen im Formatierungsstring zu unterdrücken. Das Beispiel gibt als Ergebnis `\2 \1` aus, da der gesamte String, auf den der reguläre Ausdruck zutrifft, mit dem Formatierungsstring ersetzt wird.

Beispiel 8.6 Mit `boost::regex_token_iterator` über Strings iterieren

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"\\w+"};
  boost::regex_token_iterator<std::string::iterator> it{s.begin(), s.end(),
    expr};
  boost::regex_token_iterator<std::string::iterator> end;
  while (it != end)
    std::cout << *it++ << '\n';
}
```

Boost.Regex bietet mit `boost::regex_token_iterator` eine Klasse an, um mit einem regulären Ausdruck über einen String zu iterieren. So wird im Beispiel 8.6 über die beiden Wörter im String **`s`** iteriert. Dazu wird **`it`** mit Iteratoren auf **`s`** und dem regulären Ausdruck „\w+“ initialisiert. Über den Standardkonstruktor wird ein Iterator erzeugt, der das Ende einer Iteration markiert.
Beispiel 8.6 gibt `Boost` und `Libraries` aus.

Beispiel 8.7 Mit `boost::regex_token_iterator` auf Gruppierungen zugreifen

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w)\\w+"};
  boost::regex_token_iterator<std::string::iterator> it{s.begin(), s.end(),
    expr, 1};
  boost::regex_token_iterator<std::string::iterator> end;
  while (it != end)
    std::cout << *it++ << '\n';
}
```

Sie können dem Konstruktor von `boost::regex_token_iterator` als zusätzlichen Parameter eine Zahl übergeben. Ist wie im Beispiel 8.7 1 angegeben, gibt der Iterator die erste Gruppierung im regulären Ausdruck zurück. Da „(\w)\w+“ als regulärer Ausdruck verwendet wird, gibt Beispiel 8.7 die Initialen `B` und `L` aus.

`boost::regex_token_iterator` erlaubt es, -1 anzugeben. Der reguläre Ausdruck wird dann als Trennzeichen interpretiert. Ein Iterator, der mit -1 initialisiert wurde, gibt das zurück, auf was der reguläre Ausdruck nicht zutrifft.

Beispiel 8.8 Einen regulären Ausdruck mit einem Locale verknüpfen

```
#include <boost/regex.hpp>
#include <locale>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost k\xfct\xfcphaneleri";
  boost::basic_regex<char, boost::cpp_regex_traits<char>> expr;
  expr.imbue(std::locale{"Turkish"});
  expr = "\\w+\\s\\w+";
  std::cout << std::boolalpha << boost::regex_match(s, expr) << '\n';
}
```

Im Beispiel 8.8 wird ein Locale über `imbue()` mit **`expr`** verknüpft. Dies geschieht, um den regulären Ausdruck auf den String „Boost kütüphaneleri“ anwenden zu können. Es handelt sich dabei um die türkische Übersetzung von „Die Boost-Bibliotheken“. Sollen die Umlaute im String vom regulären Ausdruck als gültige Buchstaben erkannt werden, muss das Locale gesetzt werden – andernfalls gibt `boost::regex_match()` `false` zurück. Um einen Locale vom Typ `std::locale` verwenden zu können, muss **`expr`** auf einer Regex-Klasse basieren, die mit dem Typ `boost::cpp_regex_traits` instanziiert ist. Deswegen verwendet Beispiel 8.8 nicht `boost::regex`, sondern boost::basic_regex<char, boost::cpp_regex_traits<char>>. Über den zweiten Template-Parameter von `boost::basic_regex` ist es möglich, indirekt den Parameter für `imbue()` zu definieren. Nur bei `boost::cpp_regex_traits` kann ein Locale-Objekt vom Typ `std::locale` an `imbue()` übergeben werden.

> **Anmerkung**
>
> Ersetzen Sie die Angabe „Turkish“ mit „tr_TR“, wenn Sie das Beispiel auf einem POSIX-Betriebssystem ausführen möchten. Stellen Sie außerdem sicher, dass das Locale für den türkischen Sprach- und Kulturkreis installiert ist.

Beachten Sie, dass `boost::regex` mit einem plattformabhängigen zweiten Parameter definiert ist. Unter Windows ist dies `boost::w32_regex_traits`. Damit ist es möglich, einen LCID an `imbue()` zu übergeben. Es handelt sich hierbei um eine Zahl, die unter Windows einen bestimmten Sprach- und Kulturkreis definiert. Wenn Sie plattformabhängigen Code schreiben möchten, müssen Sie wie im Beispiel 8.8 `boost::cpp_regex_traits` explizit verwenden. Alternativ können Sie das Makro `BOOST_REGEX_USE_CPP_LOCALE` definieren.

Kapitel 9

Boost.Xpressive

Boost.Xpressive bietet so wie Boost.Regex Funktionen an, um Strings mit *regulären Ausdrücken* zu durchsuchen. Reguläre Ausdrücke müssen jedoch nicht als String angegeben werden. Boost.Xpressive ermöglicht es, reguläre Ausdrücke als C++-Code zu schreiben. Das hat den Vorteil, dass bereits zur Kompilierung festgestellt werden kann, ob ein regulärer Ausdruck gültig ist oder sich zum Beispiel ein Tippfehler eingeschlichen hat. Beachten Sie, dass ausschließlich Boost.Regex in C++11 Einzug gehalten hat. Die Standardbibliothek bietet keine Unterstützung, um reguläre Ausdrücke als C++-Code angeben zu können.
Boost.Xpressive bietet mit `boost/xpressive/xpressive.hpp` eine Headerdatei an, die Sie einbinden können, um Zugriff auf die meisten Funktionen der Bibliothek zu erhalten. Für einige Funktionen müssen Sie weitere Headerdateien einbinden. Alle Definitionen der Bibliothek befinden sich im Namensraum `boost::xpressive`.

Beispiel 9.1 Strings mit `boost::xpressive::regex_match` vergleichen

```
#include <boost/xpressive/xpressive.hpp>
#include <string>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  sregex expr = sregex::compile("\\w+\\s\\w+");
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

Boost.Xpressive bietet grundsätzlich die gleichen Funktionen wie Boost.Regex. Sie befinden sich lediglich im Namensraum von Boost.Xpressive: `boost::xpressive::regex_match()` vergleicht Strings, `boost::xpressive::regex_search()` sucht in Strings, und `boost::xpressive::regex_replace()` ersetzt Zeichen in Strings. Beispiel 9.1, in dem die Funktion `boost::xpressive::regex_match()` Anwendung findet, ähnelt daher Beispiel 8.1.
Beachten Sie, dass es einen wesentlichen Unterschied zwischen Boost.Xpressive und Boost.Regex gibt. Der Typ des regulären Ausdrucks in Boost.Xpressive richtet sich nach dem Typ des Strings, der durchsucht werden soll. Da der String **s** im Beispiel 9.1 den Typ `std::string` hat, muss der reguläre Ausdruck den Typ `boost::xpressive::sregex` erhalten. Sehen Sie sich Beispiel 9.2 an, in dem der reguläre Ausdruck auf eine Zeichenkette vom Typ const char* angewandt wird.

Beispiel 9.2 `boost::xpressive::cregex` bei Zeichenketten vom Typ const char*

```
#include <boost/xpressive/xpressive.hpp>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  const char *c = "Boost Libraries";
  cregex expr = cregex::compile("\\w+\\s\\w+");
  std::cout << std::boolalpha << regex_match(c, expr) << '\n';
```

```
}
```

Für Zeichenketten vom Typ const char* verwenden Sie die Klasse `boost::xpressive::cregex`. Verwenden Sie andere Typen wie beispielsweise `std::wstring` oder const wchar_t*, greifen Sie auf `boost::xpressive::wsregex` oder `boost::xpressive::wcregex` zu.
Beachten Sie, dass Sie für reguläre Ausdrücke, die Sie als String angeben, die statische Methode `compile()` aufrufen müssen – und zwar für den Typ, den Sie für den regulären Ausdruck verwenden.
Boost.Xpressive unterstützt eine direkte Initialisierung für reguläre Ausdrücke, die als C++-Code angegeben werden. Sie müssen den regulären Ausdruck in der für Boost.Xpressive eigenen Notation angeben. Sehen Sie sich dazu Beispiel 9.3 an.

Beispiel 9.3 Ein regulärer Ausdruck als C++-Code

```
#include <boost/xpressive/xpressive.hpp>
#include <string>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  sregex expr = +_w >> _s >> +_w;
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

Der reguläre Ausdruck, der im Beispiel 9.2 als String „\w+\s\w+“ angegeben wurde, lautet nun `+_w >> _s >> +_w`. Es handelt sich dabei um den gleichen regulären Ausdruck. Auch im Beispiel 9.3 wird nach mindestens einem alphanumerischen Zeichen gesucht, gefolgt von genau einem Leerzeichen, wiederum gefolgt von mindestens einem alphanumerischen Zeichen.
Boost.Xpressive macht es möglich, reguläre Ausdrücke als C++-Code zu schreiben. Dazu bietet die Bibliothek Objekte für Zeichenklassen an. So ist zum Beispiel das Objekt **_w** gleichbedeutend mit „\w“. **_s** wiederum hat die gleiche Bedeutung wie „\s“.
Während bei regulären Ausdrücken als String Zeichenklassen wie „\w“ und „\s“ direkt hintereinander angegeben werden können, müssen Objekte wie **_w** und **_s** mit einem Operator verknüpft werden. Andernfalls würde es sich nicht um gültigen C++-Code handeln. Boost.Xpressive bietet den Operator `operator>>` an, der entsprechend im Beispiel 9.3 zum Einsatz kommt.
Um anzugeben, dass mindestens ein alphanumerisches Zeichen gefunden werden soll, wird **_w** ein Pluszeichen vorangestellt. Während die Syntax regulärer Ausdrücke vorsieht, dass eine Angabe zur Häufigkeit nach einer Zeichengruppe angegeben wird – also wie bei „\w+“ – muss das Pluszeichen vor **_w** angegeben werden. Beim Pluszeichen handelt es sich um einen unären Operator, der in C++ einem Objekt vorangestellt werden muss.
Boost.Xpressive ahmt die Regeln regulärer Ausdrücke nach, soweit sie in C++ nachgeahmt werden können. Die Bibliothek stößt dabei an Grenzen. So existiert zum Beispiel mit dem Fragezeichen ein Metazeichen, das in regulären Ausdrücken einmal oder keinmal bedeutet. Da das Fragezeichen kein gültiger Operator in C++ ist, ersetzt Boost.Xpressive es durch das Ausrufezeichen. Aus einer Angabe wie „\w?“ wird bei Boost.Xpressive `!_w`, da das Ausrufezeichen vorangestellt werden muss.
Während es bei Funktionen wie `regex_match()` fraglich ist, ob Boost.Xpressive Boost.Regex vorgezogen werden sollte, unterstützt die Bibliothek Aktionen, die mit Teilausdrücken verknüpft werden können – etwas, was Boost.Regex nicht kennt. Sehen Sie sich dazu Beispiel 9.4 an.

Beispiel 9.4 Aktionen mit Teilausdrücken verknüpfen

```
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
#include <string>
#include <iterator>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
```

```
  std::ostream_iterator<std::string> it{std::cout, "\n"};
  sregex expr = (+_w)[*boost::xpressive::ref(it) = _] >> _s >> +_w;
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

Wenn Sie Beispiel 9.4 ausführen, wird nicht mehr nur `true` für `boost::xpressive::regex_match()` ausgegeben. Es wird außerdem `Boost` ausgegeben.
Sie können Teilausdrücke mit Aktionen verknüpfen. Eine Aktion wird ausgeführt, wenn der entsprechende Teilausdruck gefunden wurde. So ist im Beispiel 9.4 für den Teilausdruck `+_w` die Aktion `*boost::xpressive::ref(it) =_` angegeben. Es handelt sich dabei um eine Lambda-Funktion. Das Objekt _ bezieht sich auf die Zeichen, die mit dem Teilausdruck gefunden wurden – in diesem Fall das erste Wort in der Variablen **`s`**. Die entsprechenden Zeichen werden dem dereferenzierten Iterator **`it`** zugewiesen. Da es sich dabei um einen Iterator vom Typ `std::ostream_iterator` handelt, der mit **`std::cout`** initialisiert ist, wird `Boost` auf die Standardausgabe ausgegeben.
Beachten Sie, dass Sie die Funktion `boost::xpressive::ref()` verwenden müssen, um den Iterator **`it`** zu kapseln. Erst dadurch wird es möglich, _ dem dereferenzierten Iterator zuzuweisen. So handelt es sich bei _ um ein Objekt, das von Boost.Xpressive im Namensraum `boost::xpressive` angeboten wird und das normalerweise nicht einem dereferenzierten Iterator vom Typ `std::ostream_iterator` zugewiesen werden kann. Da die Zuweisung außerdem nicht sofort erfolgen darf, sondern erst dann, wenn „Boost“ mit `+_w` gefunden wurde, verwandelt `boost::xpressive::ref()` die Zuweisung in eine Operation, die als *lazy* bezeichnet wird: Während der Code, der in den eckigen Klammern an `+_w` übergeben wird, gemäß den Regeln in C++ zuerst ausgeführt wird, kann die Zuweisung an den Iterator **`it`** erst dann stattfinden, wenn der reguläre Ausdruck angewandt wird. Die Angabe `*boost::xpressive::ref(it) =_` stellt eine Zuweisung dar, die nicht sofort ausgeführt wird.
Beachten Sie, dass im Beispiel 9.4 die Headerdatei `boost/xpressive/regex_actions.hpp` eingebunden wurde. Dies ist notwendig, weil Aktionen nicht automatisch über `boost/xpressive/xpressive.hpp` zur Verfügung stehen.
Boost.Xpressive unterstützt wie Boost.Regex Iteratoren, um mit regulären Ausdrücken Strings zu zerlegen. Sie greifen dazu auf die Klassen `boost::xpressive::regex_token_iterator` oder `boost::xpressive::regex_iterator` zu. Sie können außerdem wie bei Boost.Regex Locale mit einem regulären Ausdruck verknüpfen, um einen anderen als den globalen Locale zu verwenden.

Kapitel 10

Boost.Tokenizer

Die Bibliothek Boost.Tokenizer ermöglicht es, über Teilausdrücke in einem String zu iterieren. Dabei werden bestimmte Zeichen im String als Trennzeichen interpretiert.

Beispiel 10.1 Über Teilausdrücke eines Strings mit `boost::tokenizer` iterieren

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  tokenizer tok{s};
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::cout << *it << '\n';
}
```

Boost.Tokenizer definiert eine Klasse `boost::tokenizer` in der Headerdatei `boost/tokenizer.hpp`. Bei dieser Klasse handelt es sich um ein Template: Als Template-Parameter muss eine Klasse angegeben werden, die zusammenhängende Ausdrücke identifiziert. So wird im Beispiel 10.1 die Klasse `boost::char_separator` verwendet, die Leerzeichen und Interpunktionszeichen als Trennzeichen interpretiert.
Ein Tokenizer muss mit einem String vom Typ `std::string` initialisiert werden. Über die Methoden `begin()` und `end()` kann der Tokenizer wie ein Container behandelt werden. Über den Iterator wird auf Teilausdrücke des Strings zugegriffen, mit dem der Tokenizer initialisiert wurde. Wie Teilausdrücke gefunden werden, hängt von der Klasse ab, die als Template-Parameter angegeben ist.
Obiges Beispiel gibt `Boost`, `C`, `+`, `+` und `Libraries` aus, weil `boost::char_separator` standardmäßig Leerzeichen und Interpunktionszeichen als Trennzeichen interpretiert. Um diese Zeichen identifiziereren zu können, greift `boost::char_separator` auf die Funktionen `std::isspace()` und `std::ispunct()` zu. Die beiden Pluszeichen werden ausgegeben, weil Boost.Tokenizer zwischen Trennzeichen unterscheidet, die unterdrückt und ausgegeben werden sollen: Standardmäßig werden Leerzeichen unterdrückt und Interpunktionszeichen ausgegeben.

Beispiel 10.2 `boost::char_separator` initialisieren, um die Iteration anzupassen

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  boost::char_separator<char> sep{" "};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

Für den Fall, dass Interpunktionszeichen nicht als Trennzeichen interpretiert werden sollen, können Sie ein Objekt vom Typ `boost::char_separator` entsprechend initialisieren und als Parameter an den Tokenizer übergeben.
Der Konstruktor von `boost::char_separator` erwartet drei Parameter, von denen lediglich der erste angegeben werden muss. Der erste Parameter beschreibt Trennzeichen, die unterdrückt werden sollen. Für Beispiel 10.2 bedeutet das wie zuvor, dass Leerzeichen Trennzeichen sind.
Der zweite Parameter beschreibt Trennzeichen, die ausgegeben werden sollen. Wird der zweite Parameter so wie im Beispiel 10.2 nicht angegeben, ist er leer. Es gibt demnach keine Trennzeichen, die ausgegeben werden sollen. Wird Beispiel 10.2 ausgeführt, wird `Boost`, `C++` und `Libraries` ausgegeben.

Beispiel 10.3 Mit `boost::char_separator` das Standardverhalten simulieren

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  boost::char_separator<char> sep{" ", "+"};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

Wenn Sie so wie im Beispiel 10.3 als zweiten Parameter ein Pluszeichen angeben, erhalten Sie das Standardverhalten aus dem ersten Beispiel in diesem Kapitel.
Der dritte Parameter beschreibt, ob leere Teilausdrücke ausgegeben werden sollen oder nicht. Wenn zwei Trennzeichen direkt hintereinander stehen, ist der dazwischenliegende Teilausdruck leer. Standardmäßig werden diese leeren Teilausdrücke nicht ausgegeben. Sie können jedoch über den dritten Parameter das Verhalten von `boost::char_separator` ändern.

Beispiel 10.4 `boost::char_separator` initialisieren, um leere Teilausdrücke mitauszugeben

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  boost::char_separator<char> sep{" ", "+", boost::keep_empty_tokens};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

Wenn Sie Beispiel 10.4 ausführen, werden zusätzlich zwei leere Teilausdrücke ausgegeben. So befindet sich ein leerer Teilausdruck zwischen den beiden Pluszeichen und zwischen dem zweiten Pluszeichen und dem folgenden Leerzeichen.

Beispiel 10.5 Boost.Tokenizer mit Wide-Strings

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<wchar_t>,
    std::wstring::const_iterator, std::wstring> tokenizer;
  std::wstring s = L"Boost C++ Libraries";
  boost::char_separator<wchar_t> sep{L" "};
```

```
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::wcout << t << '\n';
}
```

Im Beispiel 10.5 wird über einen String vom Typ `std::wstring` iteriert. Damit der Tokenizer einen String von diesem Typ als Parameter akzeptiert, ist es notwendig, zusätzliche Template-Parameter anzugeben. Auch der Klasse `boost::char_separator` muss in diesem Fall als Template-Parameter wchar_t übergeben werden.
Neben `boost::char_separator` bietet Boost.Tokenizer zwei weitere Klassen an, um Teilausdrücke zu identifizieren.

Beispiel 10.6 `boost::escaped_list_separator` für das CSV-Format

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::escaped_list_separator<char>> tokenizer;
  std::string s = "Boost,\"C++ Libraries\"";
  tokenizer tok{s};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

Die Klasse `boost::escaped_list_separator` kann verwendet werden, um mehrere Werte zu lesen, die durch Komma getrennt sind. Dieses Format wird als CSV bezeichnet – eine Abkürzung für comma separated values. Dabei werden von `boost::escaped_list_separator` auch Anführungszeichen und Escape-Sequenzen berücksichtigt. So gibt Beispiel 10.6 `Boost` und `C++ Libraries` aus.
Eine weitere von Boost.Tokenizer zur Verfügung gestellte Klasse ist `boost::offset_separator`. Diese Klasse muss instanziiert werden. Das entsprechende Objekt muss als zweiter Parameter an den Konstruktor von `boost::tokenizer` übergeben werden.

Beispiel 10.7 Mit `boost::offset_separator` über Teilausdrücke iterieren

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::offset_separator> tokenizer;
  std::string s = "Boost_C++_Libraries";
  int offsets[] = {5, 5, 9};
  boost::offset_separator sep{offsets, offsets + 3};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

Mit `boost::offset_separator` werden die Stellen im String beschrieben, an denen Teilausdrücke enden. So gibt Beispiel 10.7 an, dass der erste Teilausdruck nach 5 Zeichen, der zweite nach weiteren 5 Zeichen und der dritte nach weiteren 6 Zeichen endet. Das Beispiel gibt `Boost`, `_C++_` und `Libraries` aus.

Kapitel II

Boost.Spirit

In diesem Kapitel lernen Sie die Bibliothek Boost.Spirit kennen. Sie verwenden Boost.Spirit, wenn Sie Parser entwickeln möchten, um textbasierte Formate zu lesen. So können Sie mit Boost.Spirit zum Beispiel Parser entwickeln, um Konfigurationsdateien zu laden. Für binäre Formate bietet sich Boost.Spirit eingeschränkt an.
Boost.Spirit vereinfacht die Entwicklung von Parsern, weil Sie das Format, das Sie lesen möchten, in Form von Regeln angeben. Regeln definieren, wie das zu lesende Format aussieht – Boost.Spirit erledigt den Rest. Sie können Boost.Spirit mit regulären Ausdrücken vergleichen, mit denen Sie nach Mustern suchen können, ohne Code entwickeln zu müssen, wie die Suche im Detail auszuführen ist.
Boost.Spirit erwartet, dass Regeln in der Parsing Expression Grammar ausgedrückt werden – kurz PEG. Die PEG ist verwandt mit der Erweiterten Backus-Naur-Form – kurz EBNF. Auch wenn Sie mit diesen Sprachen nicht vetraut sind, sollten die Beispiele in diesem Kapitel genügen, um Ihnen einen Einstieg in die Entwicklung von Parsern mit Boost.Spirit zu verschaffen.
Boost.Spirit liegt in zwei Versionen vor. Die erste Version ist als Spirit.Classic bekannt und sollte nicht mehr verwendet werden. Die aktuelle Version 2.5.2 ist die, die Sie in diesem Kapitel kennenlernen werden.
Seit der zweiten Version kann Boost.Spirit nicht nur zur Entwicklung von Parsern verwendet werden, sondern auch zur Entwicklung von Generatoren. Während Parser textbasierte Formate lesen, können Generatoren verwendet werden, um textbasierte Formate zu schreiben. Der Teil von Boost.Spirit, der für die Entwicklung von Parsern verwendet wird, wird als Spirit.Qi bezeichnet. Spirit.Karma ist der Teil der Bibliothek, der die Entwicklung von Generatoren ermöglicht. Diese Einteilung spiegelt sich in den Namensräumen wieder: Boost.Spirit definiert Klassen und Funktionen grundsätzlich in `boost::spirit`. Klassen und Funktionen speziell für die Entwicklung von Parsern finden sich im Namensraum `boost::spirit::qi`. `boost::spirit::karma` wiederum ist der Namensraum für Klassen und Funktionen zur Entwicklung von Generatoren.
Neben Spirit.Qi und Spirit.Karma bietet die Bibliothek mit Spirit.Lex auch Klassen und Funktionen zur Entwicklung von Lexern an.
Dieses Kapitel führt Sie in die Entwicklung von Parsern ein. So werden in den Beispielen vorwiegend Klassen und Funktionen aus den Namensräumen `boost::spirit` und `boost::spirit::qi` verwendet. Dazu reicht es, die Headerdatei `boost/spirit/include/qi.hpp` einzubinden.
Möchten Sie keine Master-Headerdatei wie `boost/spirit/include/qi.hpp` einbinden, können Sie auf Headerdateien in `boost/spirit/include/` einzeln verweisen. Wichtig ist, dass Sie ausschließlich auf Headerdateien in diesem Verzeichnis zugreifen. `boost/spirit/include/` ist die Schnittstelle nach außen. Headerdateien in anderen Verzeichnissen können sich in neuen Boost.Spirit-Versionen ändern, ohne dass diese Änderungen dokumentiert werden.

II.1 API

Boost.Spirit bietet mit `boost::spirit::qi::parse()` und `boost::spirit::qi::phrase_parse()` zwei Funktionen an, um Formate zu parsen.

Beispiel 11.1 `boost::spirit::qi::parse()` in Aktion

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;
```

```
int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::parse(it, s.end(), ascii::digit);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Beispiel 11.1 stellt die Funktion `boost::spirit::qi::parse()` vor. `boost::spirit::qi::parse()` erwartet zwei Iteratoren auf einen String, der geparst werden soll, sowie einen Parser. Im Beispiel wird als Parser **`boost::spirit::ascii::digit`** verwendet. Bei diesem Objekt handelt es sich um einen Parser, der von Boost.Spirit zur Verfügung gestellt wird. **`boost::spirit::ascii::digit`** zählt zu den Zeichenklassenparsern. Dies sind Parser, die Zeichen daraufhin überprüfen, ob sie zu einer bestimmten Klasse gehören. Mit **`boost::spirit::ascii::digit`** wird überprüft, ob ein Zeichen eine Ziffer zwischen 0 und 9 ist.
Im Beispiel werden Iteratoren auf einen String übergeben, der von der Standardeingabe gelesen wird. Beachten Sie, dass der Iterator auf den Anfang des Strings nicht direkt an `boost::spirit::qi::parse()` übergeben wird. Er wird in der Variablen **`it`** gespeichert, die dann an `boost::spirit::qi::parse()` übergeben wird. Der Grund ist, dass `boost::spirit::qi::parse()` den Iterator verändert. `boost::spirit::qi::parse()` gibt nicht nur ein Ergebnis vom Typ bool zurück, sondern setzt den Iterator unter Umständen neu.
Wenn Sie das Beispiel ausführen und eine Ziffer gefolgt von **Enter** eingeben, wird `true` ausgegeben. Geben Sie zwei Ziffern gefolgt von **Enter** ein, wird `true` und die zweite Ziffer ausgegeben. Geben Sie einen Buchstaben gefolgt von **Enter** ein, wird `false` und der Buchstabe ausgegeben.
Der im Beispiel 11.1 verwendete Parser **`boost::spirit::ascii::digit`** überprüft genau ein Zeichen darauf, ob es eine Ziffer ist. Ist das erste eingegebene Zeichen eine Ziffer, gibt `boost::spirit::qi::parse()` `true` zurück – andernfalls `false`. Der Rückgabewert von `boost::spirit::qi::parse()` gibt an, inwiefern ein Parser erfolgreich auf einen String angewandt werden konnte.
`boost::spirit::qi::parse()` gibt auch `true` zurück, wenn Sie mehrere Ziffern eingeben. Da der Parser **`boost::spirit::ascii::digit`** lediglich das erste eingegebene Zeichen überprüft, kann er auch in diesem Fall erfolgreich auf den String angewandt werden. Alle Zeichen ab dem zweiten werden vom Parser **`boost::spirit::ascii::digit`** jedoch ignoriert.
Um erkennen zu können, bis zu welcher Stelle im String Zeichen erfolgreich geparst wurden, verändert `boost::spirit::qi::parse()` den Iterator **`it`**. Nach einem Aufruf von `boost::spirit::qi::parse()` zeigt **`it`** auf das Zeichen nach dem letzten erfolgreich geparsten Zeichen. Geben Sie mehr als eine Ziffer ein, zeigt **`it`** auf die zweite eingegebene Ziffer. Geben Sie genaue eine Ziffer ein, ist **`it`** gleichbedeutend mit dem End-Iterator von **`s`**. Geben Sie genau einen Buchstaben ein, zeigt **`it`** auf den Buchstaben.
Beachten Sie, dass `boost::spirit::qi::parse()` Leerzeichen nicht ignoriert. Wenn Sie das obige Beispiel ausführen und ein Leerzeichen vor einer Ziffer eingeben, wird `false` ausgegeben. `boost::spirit::qi::parse()` überprüft das erste eingegebene Zeichen. Handelt es sich dabei um ein Leerzeichen, wird der Parser auf dieses angewandt. Möchten Sie Leerzeichen ignorieren, können Sie anstatt `boost::spirit::qi::parse()` die Funktion `boost::spirit::qi::phrase_parse()` aufrufen.

Beispiel 11.2 `boost::spirit::qi::phrase_parse()` in Aktion

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit, ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
```

```
}
```

`boost::spirit::qi::phrase_parse()` funktioniert wie `boost::spirit::qi::parse()`, erwartet jedoch einen zusätzlichen Parameter. Dieser wird als Skipper bezeichnet. Der Skipper ist nichts anderes als ein Parser, der Zeichen liest, die ignoriert werden sollen. Im Beispiel 11.2 wird **`boost::spirit::ascii::space`** übergeben – ein Zeichenklassenparser zum Erkennen von Leerzeichen.
Der Skipper **`boost::spirit::ascii::space`** interpretiert Leerzeichen als Trennzeichen. Wenn Sie das Beispiel ausführen und ein Leerzeichen gefolgt von einer Ziffer eingeben, wird `true` ausgegeben. Im Gegensatz zum vorherigen Beispiel wird der Parser **`boost::spirit::ascii::digit`** nicht auf das Leerzeichen angewandt, sondern auf das erste Zeichen, das kein Leerzeichen ist.
Beachten Sie, dass im obigen Beispiel beliebig viele Leerzeichen ignoriert werden. `boost::spirit::qi::phrase_parse()` gibt auch dann `true` zurück, wenn Sie mehrere Leerzeichen gefolgt von einer Ziffer eingeben. `boost::spirit::qi::phrase_parse()` verändert so wie `boost::spirit::qi::parse()` den als ersten Parameter übergebenen Iterator, so dass Sie feststellen können, bis zu welchem Zeichen der String erfolgreich geparst wurde. Im obigen Beispiel werden mögliche Leerzeichen nach einer erfolgreich geparsten Ziffer übersprungen. Wenn Sie eine Ziffer gefolgt von einem Leerzeichen gefolgt von einem Buchstaben eingeben, zeigt der Iterator direkt auf den Buchstaben. Möchten Sie, dass der Iterator auf das Leerzeichen nach der Ziffer zeigt, können Sie mit `boost::spirit::qi::skip_flag::dont_postskip` einen zusätzlichen Parameter an `boost::spirit::qi::phrase_parse()` übergeben.

Beispiel 11.3 `phrase_parse()` mit `boost::spirit::qi::skip_flag::dont_postskip`

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit, ascii::space,
    qi::skip_flag::dont_postskip);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Im Beispiel 11.3 wird `boost::spirit::qi::skip_flag::dont_postskip` an `boost::spirit::qi::phrase_parse()` übergeben, um mögliche Leerzeichen nach einer erfolgreich geparsten Ziffer nicht zu überspringen. Wenn Sie eine Ziffer gefolgt von einem Leerzeichen gefolgt von einem Buchstaben eingeben, zeigt **`it`** nach dem Aufruf von `boost::spirit::qi::phrase_parse()` auf das Leerzeichen.
Boost.Spirit unterstützt neben `boost::spirit::qi::skip_flag::dont_postskip` lediglich das Flag `boost::spirit::qi::skip_flag::postskip`. `boost::spirit::qi::skip_flag::postskip` ist der Standardwert, der gilt, wenn Sie `boost::spirit::qi::phrase_parse()` wie im vorherigen Beispiel ohne explizite Angabe eines Flags aufrufen.

Beispiel 11.4 `boost::spirit::qi::phrase_parse()` mit Wide-Strings

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::wstring s;
  std::getline(std::wcin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit, ascii::space,
    qi::skip_flag::dont_postskip);
```

```
  std::wcout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::wcout << std::wstring{it, s.end()} << '\n';
}
```

Sie können sowohl `boost::spirit::qi::parse()` als auch `boost::spirit::qi::phrase_parse()` Iteratoren auf einen Wide-String übergeben. So funktioniert Beispiel 11.4 wie das vorherige.
Boost.Spirit unterstützt auch die mit C++11 in die Standardbibliothek aufgenommen String-Typen `std::u16string` und `std::u32string`.

11.2 Parser

Nachdem Sie im vorherigen Abschnitt die Funktionen kennengelernt haben, die Boost.Spirit zum Parsen anbietet, erfahren Sie im Folgenden, wie Sie Parser entwickeln. Sie greifen dabei üblicherweise auf existierende Parser zu – wie **`boost::spirit::ascii::digit`** oder **`boost::spirit::ascii::space`**. Indem Sie von Boost.Spirit angebotene Parser kombinieren, erstellen Sie Parser, die komplexe Formate beschreiben. Sie gehen dabei ähnlich vor, wie wenn Sie reguläre Ausdrücke erstellen, die ebenfalls aus Grundbausteinen zusammengesetzt werden.

Beispiel 11.5 Ein Parser für zwei aufeinander folgende Ziffern

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit >> ascii::digit,
    ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Im Beispiel 11.5 soll eine Eingabe auf zwei Ziffern überprüft werden. `boost::spirit::qi::phrase_parse()` soll nur dann `true` zurückgeben, wenn zwei Ziffern eingegeben werden. Leerzeichen werden ignoriert.
Um Ziffern erkennen zu können, wird wie in den vorherigen Beispielen auf **`boost::spirit::ascii::digit`** zugegriffen. Weil **`boost::spirit::ascii::digit`** genau ein Zeichen überprüft, wird dieser Parser zweimal hintereinander angegeben, um eine Eingabe auf zwei Ziffern zu überprüfen. Damit **`boost::spirit::ascii::digit`** zweimal hintereinander angegeben werden kann, muss ein entsprechender Operator verwendet werden. Boost.Spirit überlädt `operator>>`, um Parser verknüpfen zu können. Mit der Angabe `ascii::digit >> ascii::digit` wird ein Parser erstellt, der einen String daraufhin überprüft, ob er zwei Ziffern enthält.
Wenn Sie das Beispiel ausführen und zwei Ziffern eingeben, wird `true` ausgegeben. Geben Sie nur eine Ziffer ein, wird `false` ausgegeben.
Beachten Sie, dass auch dann `true` ausgegeben wird, wenn Sie zwischen den beiden eingegebenen Ziffern Leerzeichen angeben. Überall dort, wo in einem Parser `>>` verwendet wird, können Zeichen stehen, die von `boost::spirit::qi::phrase_parse()` ignoriert werden. Weil im Beispiel als Skipper **`boost::spirit::ascii::space`** an `boost::spirit::qi::phrase_parse()` übergeben wird, dürfen zwischen den beiden Ziffern beliebig viele Leerzeichen stehen.
Soll der Parser die Eingabe nur dann akzeptieren, wenn die Ziffern direkt hintereinander stehen, können Sie entweder `boost::spirit::qi::parse()` verwenden oder die Direktive **`boost::spirit::qi::lexeme`**.

Beispiel 11.6 Mit **`boost::spirit::qi::lexeme`** Zeichen für Zeichen parsen

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>
```

```
using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(),
    qi::lexeme[ascii::digit >> ascii::digit], ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Im Beispiel 11.6 ist der Parser mit `qi::lexeme[ascii::digit >> ascii::digit]` angegeben. `boost::spirit::qi::phrase_parse()` gibt nur dann `true` zurück, wenn zwei Ziffern direkt hintereinander eingegeben werden – ohne trennende Leerzeichen.
`boost::spirit::qi::lexeme` ist eine von mehreren zur Verfügung stehenden Direktiven, mit denen das Verhalten von Parsern geändert werden kann. Sie verwenden **`boost::spirit::qi::lexeme`**, wenn Sie dort, wo der Operator `operator>>` angegeben ist, keine Zeichen dulden, die vom Skipper ignoriert würden.

Beispiel 11.7 Boost.Spirit-Regeln wie reguläre Ausdrücke

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), +ascii::digit, ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Im Beispiel 11.7 wird mit `+ascii::digit` ein Parser erstellt, der mindestens eine Ziffer erwartet. Diese Syntax ähnelt regulären Ausdrücken, in denen das Pluszeichen ebenfalls bedeutet, dass ein Zeichen oder eine Zeichengruppe mindestens einmal vorhanden sein muss. Wenn Sie das Beispiel ausführen und mindestens eine Ziffer eingeben, wird `true` ausgegeben. Dabei spielt keine Rolle, ob Sie die Ziffern durch Leerzeichen trennen. Soll der Parser lediglich Ziffern akzeptieren, die direkt hintereinander eingegeben werden, verwenden Sie wieder **`boost::spirit::qi::lexeme`**.

Beispiel 11.8 Nummerische Parser

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), qi::int_, ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
```

```
}
```

Beispiel 11.8 erwartet die Eingabe einer Ganzzahl. Bei **boost::spirit::qi::int_** handelt es sich um einen nummerischen Parser, der sowohl positive als auch negative Ganzzahlen lesen kann. Im Gegensatz zu einem Parser wie **boost::spirit::ascii::digit** überprüft **boost::spirit::qi::int_** nicht nur ein Zeichen. Er kann zusammenhängende Zeichen wie +1 oder -23 als Ganzzahl erkennen.
Boost.Spirit bietet neben **boost::spirit::qi::int_** weitere logische Parser an. Mit **boost::spirit::qi::float_**, **boost::spirit::qi::double_** und **boost::spirit::qi::bool_** stehen nummerische Parser zur Verfügung, die Kommazahlen und Wahrheitswerte lesen können. Mit **boost::spirit::qi::eol** existiert ein Parser, der auf ein Zeilenende überprüft. Mit **boost::spirit::qi::byte_** und **boost::spirit::qi::word** können ein oder zwei Bytes gelesen werden. **boost::spirit::qi::word** und andere binäre Parser, die mehr als ein Byte lesen, berücksichtigen automatisch die Byte-Reihenfolge – auf Englisch endianness – der Plattform. Sie können mit **boost::spirit::qi::little_word** und **boost::spirit::qi::big_word** auch auf Parser zugreifen, die eine Byte-Reihenfolge voraussetzen.

11.3 Aktionen

Die bisherigen Beispiele konnten lediglich ermitteln, ob ein Parser auf eine Eingabe erfolgreich angewandt werden konnte und bis zu welchem Zeichen eine Eingabe erfolgreich geparst wurde. Üblicherweise sollen Parser jedoch Daten laden. So soll zum Beispiel eine Zahl, die mit Hilfe des Parsers **boost::spirit::qi::int_** gelesen wurde, verarbeitet werden.

Beispiel 11.9 Aktionen mit Parsern verknüpfen

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(),
    qi::int_[([](int i){ std::cout << i << '\n'; })], ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Beispiel 11.9 greift auf **boost::spirit::qi::int_** zu, um eine Ganzzahl zu lesen. Diesmal soll die gelesene Ganzzahl zusätzlich auf die Standardausgabe ausgegeben werden. Dazu wurde **boost::spirit::qi::int_** mit einer *Aktion* verknüpft. Aktionen sind Funktionen oder Funktionsobjekte, die aufgerufen werden, wenn ein Parser erfolgreich angewandt werden konnte. Die Verknüpfung erfolgt über den Operator `operator[]`, der von **boost::spirit::qi::int_** und anderen Parsern überladen ist. So wird im obigen Beispiel eine Lambda-Funktion übergeben, die als einzigen Parameter ein int erwartet und dieses bei Aufruf auf die Standardausgabe ausgibt.
Wenn Sie Beispiel 11.9 ausführen und eine Zahl eingeben, wird diese ausgegeben.
Beachten Sie, dass sich der Typ des Parameters der Aktion nach dem Parser richtet. Während **boost::spirit::qi::int_** einen int-Wert übergibt, reicht zum Beispiel **boost::spirit::qi::float_** einen float-Wert an die Aktion weiter.

Beispiel 11.10 Boost.Spirit mit Boost.Phoenix

```
#define BOOST_SPIRIT_USE_PHOENIX_V3
#include <boost/spirit/include/qi.hpp>
#include <boost/phoenix/phoenix.hpp>
#include <string>
#include <iostream>
```

```
using namespace boost::spirit;
using boost::phoenix::ref;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  int i;
  bool match = qi::phrase_parse(it, s.end(), qi::int_[ref(i) = qi::_1],
    ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (match)
    std::cout << i << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Beispiel 11.10 verwendet Boost.Phoenix, um den von **`boost::spirit::qi::int_`** geparsten int-Wert in einer Variablen **`i`** zu speichern. Gibt `boost::spirit::qi::phrase_parse()` `true` zurück, wird **`i`** auf die Standardausgabe ausgegeben.
Beachten Sie, dass im obigen Beispiel das Makro `BOOST_SPIRIT_USE_PHOENIX_V3` definiert wird, bevor die Headerdatei von Boost.Spirit eingebunden wird. Mit diesem Makro wird die dritte und aktuelle Boost.Phoenix-Version gewählt. Das Makro ist insofern entscheidend, als dass Boost.Phoenix aus Boost.Spirit hervorgegangen ist und Boost.Spirit die zweite Version von Boost.Phoenix enthält. Wenn Sie `BOOST_SPIRIT_USE_PHOENIX_V3` nicht definieren, wird über Boost.Spirit die zweite Version von Boost.Phoenix und über die Headerdatei `boost/phoenix/phoenix.hpp` die dritte Version eingebunden, was zu einem Compilerfehler führt.
Beachten Sie außerdem, wie die Lambda-Funktion im Detail definiert ist. Über `boost::phoenix::ref()` wird eine Referenz auf die Variable **`i`** erstellt. Der Platzhalter **`_1`** stammt jedoch nicht aus Boost.Phoenix, sondern aus Boost.Spirit. Das ist wichtig, weil über **`boost::spirit::qi::_1`** der geparste Wert in dem Typ zur Verfügung gestellt wird, der üblicherweise erwartet wird – im obigen Beispiel ein int. Würde die Lambda-Funktion auf den Platzhalter **`boost::phoenix::placeholders::arg1`** zugreifen, gäbe es einen Compilerfehler. **`boost::phoenix::placeholders::arg1`** würde keinem int-Wert entsprechen, sondern einem anderen Typen aus Boost.Spirit, von dem der int-Wert erhalten werden müsste.
Die Dokumentation von Boost.Spirit enthält eine Übersicht über Hilfsmittel, die das Zusammenspiel mit Boost.Phoenix erleichtern.

II.4 Attribute

Aktionen sind eine Möglichkeit, geparste Werte zu verarbeiten. Eine andere Möglichkeit ist es, Objekte an `boost::spirit::qi::parse()` oder `boost::spirit::qi::phrase_parse()` zu übergeben, in denen die geparsten Werte gespeichert werden. Diese Objekte werden Attribute genannt. Entscheidend ist, dass der Typ der Attribute zum Parser passt.
Genaugenommen sind Sie mit Attributen bereits im vorherigen Abschnitt in Berührung gekommen. Der Parameter, der an Aktionen übergeben wird, ist ein Attribut. Jeder Parser besitzt ein Attribut. So hat zum Beispiel das Attribut des Parsers **`boost::spirit::qi::int_`** den Typ int. Im Folgenden werden Attribute jedoch nicht als Parameter an Funktionen übergeben. Geparste Daten werden in Attributen gespeichert und stehen nach einem Aufruf von `boost::spirit::qi::parse()` oder `boost::spirit::qi::phrase_parse()` zur Weiterverarbeitung zur Verfügung.

Beispiel 11.11 Einen int-Wert in einem Attribut speichern

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
```

```
  std::getline(std::cin, s);
  auto it = s.begin();
  int i;
  if (qi::phrase_parse(it, s.end(), qi::int_, ascii::space, i))
    std::cout << i << '\n';
}
```

Im Beispiel 11.11 kommt ein Parser vom Typ **`boost::spirit::qi::int_`** zum Einsatz. Der geparste int-Wert soll in der Variablen **`i`** gespeichert werden. Dies erfolgt nicht über eine Aktion. Stattdessen wird **`i`** als zusätzlicher Parameter an `boost::spirit::qi::phrase_parse()` übergeben. **`i`** wird dadurch zu einem Attribut des Parsers und automatisch auf den Wert gesetzt, der vom Parser gefunden wird.
Wenn Sie das obige Beispiel ausführen und eine Zahl eingeben, wird diese auf die Standardausgabe ausgegeben.

Beispiel 11.12 Mehrere int-Werte in einem Attribut speichern

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  std::vector<int> v;
  if (qi::phrase_parse(it, s.end(), qi::int_ % ',', ascii::space, v))
  {
    std::ostream_iterator<int> out{std::cout, ";"};
    std::copy(v.begin(), v.end(), out);
  }
}
```

Im Beispiel 11.12 wird ein Parser verwendet, der mit `qi::int_ % ','` definiert ist. Dieser Parser erlaubt die Eingabe beliebig vieler Ganzzahlen, die durch Kommas getrennt werden müssen. Wie gewohnt spielt es keine Rolle, an welchen Stellen wie viele Leerzeichen angegeben werden – sie werden im obigen Beispiel ignoriert. Weil der Parser mehrere int-Werte zurückgeben kann, muss das Attribut einen Typ besitzen, der mehrere int-Werte speichern kann. Im Beispiel wird ein Vektor übergeben. Wenn Sie das Beispiel ausführen und mehrere Ganzzahlen durch Kommas getrennt eingeben, werden sie durch Semikolons getrennt auf die Standardausgabe ausgegeben.
Sie können anstelle des Vektors auch einen anderen Container wie zum Beispiel `std::list` verwenden.
Die Dokumentation von Boost.Spirit enthält eine Übersicht, welcher Attribut-Typ für welchen Operator verwendet werden muss.

11.5 Regeln

In Boost.Spirit bestehen Parser aus Regeln. Da Regeln üblicherweise auf von Boost.Spirit angebotenen Parsern basieren, fällt eine klare Unterscheidung schwer. So kann **`boost::spirit::ascii::digit`** sowohl Parser als auch Regel sein. Grundsätzlich werden unter Regeln jedoch komplexere Ausdrücke wie `qi::int_ % ','` verstanden.
In allen bisherigen Beispielen wurden Regeln direkt an `boost::spirit::qi::parse()` oder `boost::spirit::qi::phrase_parse()` übergeben. Mit `boost::spirit::qi::rule` bietet Boost.Spirit eine Klasse an, die zur Regeldefinition verwendet werden kann. Sollen Ihre Regeln zum Beispiel Eigenschaften einer Klasse sein, müssen Sie `boost::spirit::qi::rule` verwenden.

Beispiel 11.13 Regeln mit `boost::spirit::qi::rule` definieren

```
#include <boost/spirit/include/qi.hpp>
```

```
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  qi::rule<std::string::iterator, std::vector<int>(),
    ascii::space_type> values = qi::int_ % ',';
  std::vector<int> v;
  if (qi::phrase_parse(it, s.end(), values, ascii::space, v))
  {
    std::ostream_iterator<int> out{std::cout, ";"};
    std::copy(v.begin(), v.end(), out);
  }
}
```

Beispiel 11.13 funktioniert wie Beispiel 11.12: Wenn Sie das Programm ausführen und mehrere Ganzzahlen durch Kommas getrennt eingeben, werden diese durch Semikolons getrennt ausgegeben. Im Gegensatz zum vorherigen Beispiel wird der Parser nicht direkt an `boost::spirit::qi::phrase_parse()` übergeben, sondern über eine Variable vom Typ `boost::spirit::qi::rule`.
`boost::spirit::qi::rule` ist ein Template. Sie müssen als einzigen Parameter den Iteratortyp des Strings angeben, der geparst werden soll. Im Beispiel sind zwei zusätzliche Parameter angegeben, die optional sind.
Der zweite Template-Parameter std::vector<int>() ist die Signatur einer Funktion. Die Funktion gibt einen Vektor vom Typ std::vector<int> zurück und erwartet keinen Parameter. Dieser Template-Parameter besagt, dass der Typ des Attributs, das geparst wird, ein int-Vektor ist.
Der dritte Template-Parameter ist der Typ des Skippers, der von `boost::spirit::qi::phrase_parse()` verwendet wird, um Zeichen zu überspringen. Im Beispiel wird **`boost::spirit::ascii::space`** an `boost::spirit::qi::phrase_parse()` übergeben, um Leerzeichen zu ignorieren. Der Typ dieses Skippers kann über `boost::spirit::ascii::space_type` erhalten werden und wird entsprechend als dritter Parameter an `boost::spirit::qi::rule` übergeben.

> **Anmerkung**
>
> Wenn Sie plattformunabhängigen Code schreiben möchten und in einer C++11-Entwicklungsumgebung arbeiten, sollten Sie `boost::spirit::qi::rule` dem Schlüsselwort `auto` vorziehen. Wenn Sie die Variable **`values`** mit `auto` definieren, funktioniert das Beispiel mit GCC und Clang wie erwartet. Mit Visual C++ 2013 wird jedoch nur die erste eingegebene Zahl geparst und auf die Standardausgabe ausgegeben.

Beispiel 11.14 Regeln verschachteln

```
#include <boost/spirit/include/qi.hpp>
#include <boost/variant.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::spirit;

struct print : public boost::static_visitor<>
{
  template <typename T>
```

```
  void operator()(T t) const
  {
    std::cout << std::boolalpha << t << ';';
  }
};

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  qi::rule<std::string::iterator, boost::variant<int, bool>(),
    ascii::space_type> value = qi::int_ | qi::bool_;
  qi::rule<std::string::iterator, std::vector<boost::variant<int, bool>>(),
    ascii::space_type> values = value % ',';
  std::vector<boost::variant<int, bool>> v;
  if (qi::phrase_parse(it, s.end(), values, ascii::space, v))
  {
    for (const auto &elem : v)
      boost::apply_visitor(print{}, elem);
  }
}
```

Im Beispiel 11.14 kommen zwei Regeln zum Einsatz, von denen die eine auf die andere verweist: Während **`values`** mit `value % ','` definiert ist, lautet die Definition von **`value`** `qi::int_ | qi::bool_`. **`values`** gibt an, dass beliebig viele Werte mit Kommas getrennt angegeben werden können. **`value`** wiederum definiert einen Wert als Ganzzahl oder Wahrheitswert. Zusammengenommen bedeutet dies, dass Ganzzahlen und Wahrheitswerte durch Kommas getrennt in einer beliebigen Reihenfolge geparst werden können.
Um beliebig viele Werte speichern zu können, wird ein Container vom Typ `std::vector` bereitgestellt. Da die Werte entweder eine Ganzzahl vom Typ int oder ein Wahrheitswert vom Typ bool sind, wird ein entsprechender Typ benötigt, der int und bool vereint. Wie in der Übersicht zu Attribut-Typen und Operatoren erkennbar muss dazu auf die Klasse `boost::variant` von Boost.Variant zugegriffen werden.
Wenn Sie das Beispiel ausführen und Ganzzahlen und Wahrheitswerte durch Kommas getrennt eingeben, werden diese durch Semikolons getrennt auf die Standardausgabe ausgegeben. Dies geschieht mit Hilfe der Funktion `boost::apply_visitor()`, die von Boost.Variant angeboten wird. Diese Funktion erwartet einen Visitor. Zu diesem Zweck wurde die Klasse `print` definiert.
Beachten Sie, dass Sie Wahrheitswerte mit **`true`** und **`false`** eingeben müssen.

II.6 Grammatik

Wenn Sie komplexere Formate parsen möchten und mehrere Regeln erstellen, die sich aufeinander beziehen, können Sie diese gruppieren. Boost.Spirit bietet hierfür die Klasse `boost::spirit::qi::grammar` an.

Beispiel 11.15 Regeln in einer Grammatik gruppieren

```
#include <boost/spirit/include/qi.hpp>
#include <boost/variant.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::spirit;

template <typename Iterator, typename Skipper>
struct my_grammar : qi::grammar<Iterator,
  std::vector<boost::variant<int, bool>>(), Skipper>
{
  my_grammar() : my_grammar::base_type{values}
  {
    value = qi::int_ | qi::bool_;
    values = value % ',';
  }
```

```
  qi::rule<Iterator, boost::variant<int, bool>(), Skipper> value;
  qi::rule<Iterator, std::vector<boost::variant<int, bool>>(), Skipper>
    values;
};

struct print : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const
  {
    std::cout << std::boolalpha << t << ';';
  }
};

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  my_grammar<std::string::iterator, ascii::space_type> g;
  std::vector<boost::variant<int, bool>> v;
  if (qi::phrase_parse(it, s.end(), g, ascii::space, v))
  {
    for (const auto &elem : v)
      boost::apply_visitor(print{}, elem);
  }
}
```

Beispiel 11.15 funktioniert wie Beispiel 11.14: Sie können Ganzzahlen und Wahrheitswerte durch Kommas getrennt in einer beliebigen Reihenfolge angeben, die vom Programm durch Semikolons getrennt ausgegeben werden. So greift auch dieses Beispiel auf die beiden Regeln **`value`** und **`values`** zu. In diesem Beispiel sind die Regeln zu einer Grammatik zusammengefasst. So ist eine Klasse `my_grammar` definiert, die von `boost::spirit::qi::grammar` abgeleitet ist.
Sowohl `my_grammar` als auch `boost::spirit::qi::grammar` sind Templates. `my_grammar` reicht die Template-Parameter an `boost::spirit::qi::grammar` weiter. Die Template-Parameter, die `boost::spirit::qi::grammar` erwartet, sind gleichbedeutend mit denjenigen von `boost::spirit::qi::rule`. So muss `boost::spirit::qi::grammar` mindestens ein Iteratortyp für einen String übergeben werden. Optional kann die Signatur einer Funktion übergeben werden, die den Typ des Attributs angibt, und der Typ eines Skippers.
In der Klasse `my_grammar` wird auf `boost::spirit::qi::rule` zugegriffen, um die Regeln **`value`** und **`values`** zu erstellen. Die Regeln sind als Eigenschaften definiert und werden im Konstruktor von `my_grammar` initialisiert.
Beachten Sie, dass die äußere Regel über base_type an den Konstruktor der Elternklasse übergeben werden muss. So weiß Boost.Spirit, welche Regel der Einstiegspunkt in die Grammatik ist.
Ist eine Grammatik definiert, kann sie wie ein Parser verwendet werden. So wird `my_grammar` in `main()` instanziiert und das entsprechende Objekt **`g`** an `boost::spirit::qi::phrase_parse()` übergeben.

Beispiel 11.16 Geparste Werte in Strukturen speichern

```
#include <boost/spirit/include/qi.hpp>
#include <boost/variant.hpp>
#include <boost/fusion/include/adapt_struct.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::spirit;

typedef boost::variant<int, bool> int_or_bool;

struct int_or_bool_values
{
  int_or_bool first;
  std::vector<int_or_bool> others;
```

```
};

BOOST_FUSION_ADAPT_STRUCT(
  int_or_bool_values,
  (int_or_bool, first)
  (std::vector<int_or_bool>, others)
)

template <typename Iterator, typename Skipper>
struct my_grammar : qi::grammar<Iterator, int_or_bool_values(), Skipper>
{
  my_grammar() : my_grammar::base_type{values}
  {
    value = qi::int_ | qi::bool_;
    values = value >> ',' >> value % ',';
  }

  qi::rule<Iterator, int_or_bool(), Skipper> value;
  qi::rule<Iterator, int_or_bool_values(), Skipper> values;
};

struct print : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const
  {
    std::cout << std::boolalpha << t << ';';
  }
};

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  my_grammar<std::string::iterator, ascii::space_type> g;
  int_or_bool_values v;
  if (qi::phrase_parse(it, s.end(), g, ascii::space, v))
  {
    print p;
    boost::apply_visitor(p, v.first);
    for (const auto &elem : v.others)
      boost::apply_visitor(p, elem);
  }
}
```

Beispiel 11.16 basiert auf dem vorherigen, erwartet jedoch die Eingabe mindestens zweier Werte. Die Regel in **`values`** lautet `value >> ',' >> value % ','`.
Die erste Komponente in **`values`** ist `value`, die zweite `value % ','`. Der zuerst geparste Wert muss in einem Objekt vom Typ `boost::variant` gespeichert werden. Die von der zweiten Komponente geparsten Werte müssen in einem Container gespeichert werden. Mit `int_or_bool_values` enthält das Beispiel eine entsprechende Struktur, die den beiden Komponenten der Regel **`values`** entspricht.
Um `int_or_bool_values` mit Boost.Spirit verwenden zu können, wird auf das Makro `BOOST_FUSION_ADAPT_STRUCT` zugegriffen, das von Boost.Phoenix zur Verfügung gestellt wird. Über dieses Makro kann die Struktur `int_or_bool_values` wie ein Tuple behandelt werden – ein Tuple mit zwei Werten vom Typ int_or_bool und std::vector<int_or_bool>. Weil dieses Tuple die richtige Anzahl an Werten mit den richtigen Typen besitzt, kann **`values`** mit einer Signatur `int_or_bool_values()` definiert werden. **`values`** wird den ersten geparsten Wert in der Eigenschaft **`first`** und die weiteren geparsten Werte in der Eigenschaft **`others`** speichern.
`boost::spirit::qi::phrase_parse()` wird als Attribut ein Objekt vom Typ `int_or_bool_values` übergeben. Wenn Sie das Programm ausführen und mindestens zwei Ganzzahlen oder Wahrheitswerte durch Kommas getrennt eingeben, werden diese im Attribut gespeichert und anschließend auf die Standardausgabe ausgegeben.

Anmerkung

Der Parser wurde im Beispiel im Vergleich zum vorherigen absichtlich abgewandelt und erwartet aus gutem Grund mindestens zwei Werte. Würde **`values`** mit `value % ','` definiert sein, würde `int_or_bool_values` lediglich eine Eigenschaft besitzen. So würden alle Werte wie im vorherigen Beispiel in einem Vektor gespeichert werden können. In diesem Fall würde `int_or_bool_values` einem Tuple mit lediglich einem Wert entsprechen – etwas, was Boost.Spirit nicht unterstützt. Strukturen mit lediglich einer Eigenschaft führen zu einem Compilerfehler. Es existieren mehrere Workarounds für dieses Problem.

Teil III

Container

Container sind eine der nützlichsten Datenstrukturen in C++. So bietet die Standardbibliothek aus gutem Grund zahlreiche Container an. Die Boost-Bibliotheken stellen noch mehr zur Verfügung.

- Boost.MultiIndex bietet nicht einfach einen Container an, sondern geht einen Schritt weiter: Die Container dieser Bibliothek können die von anderen Containern bekannten Schnittstellen gleichzeitig unterstützen. Container von Boost.MultiIndex sind gewissermaßen aus mehreren Containern verschmolzen und bieten all ihre Vorteile in einem Container vereint an.
- Boost.Bimap basiert auf Boost.MultiIndex. Die Bibliothek stellt einen Container ähnlich `std::unordered_map` zur Verfügung. Im Container gespeicherte Elemente können jedoch von beiden Seiten nachgeschlagen werden. Was Schlüssel und Wert ist, hängt davon ab, von welcher Seite auf den Container zugegriffen wird.
- Boost.Array und Boost.Unordered bieten mit `boost::array`, `boost::unordered_set` und `boost::unordered_map` Container an, die mit C++11 in die Standardbibliothek aufgenommen wurden.
- Boost.CircularBuffer bietet einen Ringspeicher an. Wichtigste Eigenschaft: Wird einem vollen Ringspeicher ein Element hinzugefügt, wird das erste Element im Ringspeicher überschrieben.
- Boost.Heap bietet zahlreiche Varianten von Priority-Queues an – also Klassen, die `std::priority_queue` ähneln.
- Alle Container aus der Standardbibliothek kopieren oder – seit C++11 – verschieben Objekte, wenn diese einem Container hinzugefügt werden. Mit Boost.Intrusive können Container verwendet werden, die ein Objekt weder kopieren noch verschieben. Damit das Originalobjekt zum Beispiel einer intrusiven Liste hinzugefügt werden kann, muss dessen Typ jedoch bestimmte Anforderungen erfüllen.
- Boost.MultiArray ist bemüht, den Umgang mit mehrdimensionalen Arrays zu vereinfachen. So können zum Beispiel Teilausschnitte eines multidimensionalen Arrays wie eigenständige Arrays behandelt werden.
- Boost.Container ist eine Bibliothek, die die gleichen Container wie die Standardbibliothek anbietet. Ihr Einsatz kann zum Beispiel in Programmen sinnvoll sein, die für mehrere Plattformen angeboten werden sollen, um Probleme mit möglicherweise unterschiedlichen Implementationen der Container der Standardbibliothek zu vermeiden.

Kapitel 12

Boost.MultiIndex

Die Bibliothek Boost.MultiIndex ermöglicht es, Container zu definieren, die beliebig viele Schnittstellen unterstützen. Während zum Beispiel `std::vector` die für Vektoren typische Schnittstelle anbietet, über die per Index ein direkter Zugriff auf Elemente möglich ist, und während zum Beispiel `std::set` eine Schnittstelle anbietet, über die Elemente sortiert dargestellt werden, kann mit Boost.MultiIndex ein Container definiert werden, der beide Schnittstellen unterstützt. So könnte auf Elemente in diesem Container sowohl direkt per Index als auch über eine sortierte Sichtweise zugegriffen werden.
Boost.MultiIndex bietet sich an, wenn auf Elemente unterschiedlich zugegriffen werden muss und sie deswegen in mehreren Containern gespeichert werden müssten. Anstatt Elemente sowohl in einem Vektor als auch in einem Set zu speichern und die Container laufend zu synchronisieren, kann ein Container mit Boost.MultiIndex definiert werden, der sowohl die von Vektoren als auch die von Sets bekannten Schnittstellen anbietet.
Boost.MultiIndex ist auch von Vorteil, wenn nicht nur mehrere Schnittstellen benötigt werden, sondern der Zugriff auf Elemente von unterschiedlichen Eigenschaften der Elemente abhängt. Sehen Sie sich dazu Beispiel 12.1 an, in dem Tiere sowohl über ihren Namen als auch über die Anzahl der Beine nachgeschlagen werden können. Ohne Boost.MultiIndex müssten zwei Hash-Container verwendet werden, wobei der eine Hash-Container Tiere nach Namen und der andere Hash-Container Tiere nach Anzahl der Beine speichern müsste.

Beispiel 12.1 `boost::multi_index::multi_index_container` in Aktion

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;
```

```
int main()
{
  animal_multi animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  std::cout << animals.count("cat") << '\n';

  const animal_multi::nth_index<1>::type &legs_index = animals.get<1>();
  std::cout << legs_index.count(8) << '\n';
}
```

Wenn Sie Boost.MultiIndex verwenden, ist der erste Schritt, einen neuen Container zu definieren. So müssen Sie entscheiden, welche Schnittstellen Ihr neuer Container unterstützen soll und auf welche Eigenschaften von Elementen Schnittstellen zugreifen sollen.
Eine Klasse, die bei jeder Containerdefinition benötigt wird, ist `boost::multi_index::multi_index_container`. Sie ist in der Headerdatei `boost/multi_index_container.hpp` definiert. Bei `boost::multi_index::multi_index_container` handelt es sich um ein Template, dem mindestens zwei Parameter übergeben werden müssen. Der erste Parameter ist der Typ, der im Container gespeichert werden soll – im Beispiel 12.1 `animal`. Der zweite Parameter wird verwendet, um die verschiedenen Indizes festzulegen, die vom Container zur Verfügung gestellt werden sollen.
Der entscheidende Vorteil, den Container auf Basis von Boost.MultiIndex bieten, ist, dass auf Elemente über unterschiedliche Schnittstellen zugegriffen werden kann. Bei einer Containerdefinition muss angegeben werden, wie viele und welche Schnittstellen ein Container anbieten soll. Da für Beispiel 12.1 ein Container benötigt wird, mit dem sowohl über den Namen als auch über die Anzahl der Beine nach Tieren gesucht werden kann, werden zwei Schnittstellen definiert. Boost.MultiIndex nennt diese Schnittstellen Indizes – daher der Name dieser Bibliothek.
Die Definition von Schnittstellen erfolgt über die Klasse `boost::multi_index::indexed_by`. Auch bei dieser Klasse handelt es sich um ein Template, dem mehrere Parameter übergeben werden können, wobei jeder Parameter eine Schnittstelle definiert. Im Beispiel 12.1 werden zwei Schnittstellen vom Typ `boost::multi_index::hashed_non_unique` definiert, deren Definition sich in der Headerdatei `boost/multi_index/hashed_index.hpp` befindet. Diese Schnittstelle wird verwendet, wenn sich ein Container ähnlich wie ein `std::unordered_set` verhalten soll und Elemente über einen Hashwert gefunden werden sollen.
Auch die Klasse `boost::multi_index::hashed_non_unique` ist ein Template. Sie erwartet als einzigen Parameter eine Klasse, die sie zur Berechnung von Hashwerten verwenden kann. Da die beiden Schnittstellen des Containers den Zugriff auf Tiere über Namen und Beine ermöglichen sollen, soll die eine Schnittstelle Hashwerte für Namen und die andere Schnittstelle Hashwerte für Beine errechnen.
Boost.MultiIndex bietet für den Zugriff auf eine Eigenschaft eine Hilfsklasse `boost::multi_index::member` an. Sie ist in der Headerdatei `boost/multi_index/member.hpp` definiert. Da es sich auch bei dieser Klasse um ein Template handelt, müssen so wie im Beispiel 12.1 mehrere Parameter angegeben werden, damit `boost::multi_index::member` weiß, auf welche Eigenschaft von `animal` zugegriffen werden soll und welchen Typ die Eigenschaft hat.
Auch wenn die Klassendefinition von `animal_multi` auf den ersten Blick kompliziert aussieht: Die Klasse ähnelt einer Map. Der Name und die Anzahl der Beine eines Tiers können als Schlüssel/Wert-Paar betrachtet werden. Ein Vorteil des Containers `animal_multi` gegenüber einer Map wie `std::unordered_map` ist es, dass sowohl der Name als auch die Anzahl der Beine der Schlüssel sein kann, um nach einem Tier zu suchen. `animal_multi` unterstützt zwei Schnittstellen – und je nach Schnittstelle findet der Aufruf über den Namen oder über die Beine statt. Was der Schlüssel und was der Wert ist, hängt von der Schnittstelle ab, über die auf den Container zugegriffen wird.
Wenn Sie auf einen MultiIndex-Container zugreifen, müssen Sie sich entscheiden, welche Schnittstelle Sie für den Zugriff verwenden möchten. Wenn wie im Beispiel 12.1 per `insert()` oder `count()` direkt auf **`animals`** zugegriffen wird, wird automatisch die erste Schnittstelle verwendet – in diesem Fall also der Hash-Container für die Eigenschaft **`name`**. Möchten Sie über eine andere als die erste Schnittstelle auf einen MultiIndex-Container zugreifen, müssen Sie sie explizit auswählen.
Schnittstellen sind durchnummeriert, wobei die erste Schnittstelle den Index 0 besitzt. Wenn Sie wie im Beispiel 12.1 auf die zweite Schnittstelle zugreifen möchten, können Sie dies über `get()` machen. Sie müssen dieser Methode, die ein Template ist, den Index der gewünschten Schnittstelle als Template-Parameter übergeben.

Der Rückgabewert von `get()` sieht kompliziert aus: Sie greifen auf eine Klasse im MultiIndex-Container zu, die `nth_index` heißt. Weil es sich hier wiederum um ein Template handelt, müssen Sie den Index der Schnittstelle, die Sie verwenden wollen, als Template-Parameter angeben. Dieser Index muss der gleiche sein wie der, den Sie als Template-Parameter an `get()` übergeben haben. Wenn Sie dann auf die Typdefinition type der Klasse `nth_index` zugreifen, haben Sie es geschafft. type repräsentiert den Typ der entsprechenden Schnittstelle. In den nachfolgenden Beispielen wird das Schlüsselwort `auto` verwendet, um den Code lesbarer zu machen.
Sie müssen zwar nicht wissen, wie die Typdefinition einer Schnittstelle exakt aussieht, weil sie über `nth_index` und type automatisch abgeleitet werden kann. Sie sollten aber wissen, auf welche Art von Schnittstelle Sie zugreifen. Das sollte sich aber von selbst verstehen: Da Sie den Index der Schnittstelle an `get()` und `nth_index` übergeben und in der Containerdefinition nachsehen können, in welcher Reihenfolge Schnittstellen definiert sind, wissen Sie, welche Art von Schnittstelle Ihnen zur Verfügung steht. So gilt für obiges Beispiel: **`legs_index`** ist ebenfalls eine Hash-Schnittstelle, nur dass in diesem Fall nicht über den Namen, sondern über die Anzahl der Beine nach Tieren gesucht wird.
Wenn in einem MultiIndex-Container wie im Beispiel 12.1 Daten wie Name und Beine je nach Schnittstelle Schlüssel sein können, dürfen sie nicht beliebig geändert werden können. Wenn zum Beispiel nach einem Tier mit einem bestimmten Namen gesucht wird und die Anzahl der Beine des Tiers geändert werden könnte, wüsste die andere Schnittstelle, die die Anzahl der Beine als Schlüssel verwendet, nicht, dass einer der Schlüssel geändert wurde und ein neuer Hashwert gebildet werden muss.
So wie es nicht möglich ist, den Schlüssel in einem Container vom Typ `std::unordered_map` zu ändern, können Daten in einem MultiIndex-Container nicht geändert werden. Grundsätzlich sind alle Daten, die in einem MultiIndex-Container gespeichert wurden, konstant. Das schließt Eigenschaften ein, die von keiner Schnittstelle verwendet werden. Selbst wenn es keine Schnittstelle geben würde, die auf **`legs`** zugreift, könnte **`legs`** nicht verändert werden.
Damit Elemente nicht gelöscht und in geänderter Form wieder gespeichert werden müssen, bietet Boost.MultiIndex Methoden an, über die Elemente im Container direkt geändert werden können. Da diese Methoden für den MultiIndex-Container aufgerufen werden und Elemente im MultiIndex-Container nicht über einen direkten Zugriff auf diese geändert werden, ist diese Art der Datenänderung sicher. So werden in diesem Fall sämtliche Schnittstellen benachrichtigt und können zum Beispiel Hashwerte neu berechnen.

Beispiel 12.2 Elemente in einem MultiIndex-Container mit `modify()` ändern

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
```

```
  animal_multi animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  auto &legs_index = animals.get<1>();
  auto it = legs_index.find(4);
  legs_index.modify(it, [](animal &a){ a.name = "dog"; });

  std::cout << animals.count("dog") << '\n';
}
```

Die Methode `modify()` wird von allen Schnittstellen in Boost.MultiIndex unterstützt. Sie kann immer direkt für einen MultiIndex-Container aufgerufen werden. Ihr muss als erster Parameter ein Iterator auf ein Element im Container übergeben werden, das geändert werden soll. Der zweite Parameter ist eine Funktion oder ein Funktionsobjekt, dem beim Aufruf als einziger Parameter ein Element aus dem MultiIndex-Container übergeben wird. In dieser Funktion kann das Element beliebig geändert werden.
Bisher haben Sie lediglich eine einzige Schnittstelle kennengelernt: Mit `boost::multi_index::hashed_non_unique` werden Elemente über einen Hashwert nachgeschlagen, wobei die Hashwerte nicht eindeutig sein müssen. Möchten Sie verhindern, dass mehrere Elemente mit dem gleichen Hashwert im Container gespeichert werden, können Sie `boost::multi_index::hashed_unique` verwenden.
Beachten Sie, dass Container Elemente nicht speichern können, wenn nicht alle Anforderungen aller Schnittstellen des Containers erfüllt sind. Wenn eine Schnittstelle ein Element nicht akzeptiert, hilft es nicht, wenn andere Schnittstellen dieses akzeptieren.

Beispiel 12.3 Ein MultiIndex-Container mit `boost::multi_index::hashed_unique`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
```

```
  animals.insert({"dog", 4});

  auto &legs_index = animals.get<1>();
  std::cout << legs_index.count(4) << '\n';
}
```

Der MultiIndex-Container im Beispiel 12.3 verwendet als zweite Schnittstelle die Klasse `boost::multi_index::hashed_unique`. Das bedeutet, dass keine zwei Tiere die gleiche Anzahl an Beinen haben dürfen. Dann würde für beide Tiere der gleiche Hashwert errechnet werden – und der muss, was die zweite Schnittstelle betrifft, einmalig sein.
Weil im Beispiel versucht wird, einen Hund zu speichern, der wie die Katze vier Beine hat, werden die Anforderungen der zweiten Schnittstelle verletzt. Der Hund kann dem MultiIndex-Container nicht hinzugefügt werden und wird ignoriert. Wenn Sie das Beispiel ausführen, wird `1` ausgegeben, da der Container genau ein Tier mit vier Beinen speichert – die Katze.

Beispiel 12.4 Die Schnittstellen `sequenced`, `ordered_non_unique` und `random_access`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/random_access_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    sequenced<>,
    ordered_non_unique<
      member<
        animal, int, &animal::legs
      >
    >,
    random_access<>
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.push_back({"cat", 4});
  animals.push_back({"shark", 0});
  animals.push_back({"spider", 8});

  auto &legs_index = animals.get<1>();
  auto it = legs_index.lower_bound(4);
  auto end = legs_index.upper_bound(8);
  for (; it != end; ++it)
    std::cout << it->name << '\n';

  const auto &rand_index = animals.get<2>();
  std::cout << rand_index[0].name << '\n';
}
```

Beispiel 12.4 stellt Ihnen die anderen drei Schnittstellen vor, die Boost.MultiIndex anbietet: `boost::multi_index::sequenced`, `boost::multi_index::ordered_non_unique` und `boost::multi_index::random_access`.
Über die Schnittstelle `boost::multi_index::sequenced` können Sie einen MultiIndex-Container wie eine Liste behandeln – also ähnlich wie `std::list`. Aus Sicht von `boost::multi_index::sequenced` werden Elemente genau in der Reihenfolge gespeichert, in der sie dem Container hinzugefügt werden.
Wenn Sie die Schnittstelle `boost::multi_index::ordered_non_unique` verwenden, werden Elemente automatisch sortiert. Für diese Schnittstelle müssen Sie bei der Definition des Containers angeben, wie die Sortierung erfolgen soll. So wird im Beispiel 12.4 angegeben, dass Objekte vom Typ `animal` über die Anzahl der Beine sortiert werden sollen. Dazu wird auf die bereits bekannte Hilfsklasse `boost::multi_index::member` zugegriffen.
Da `boost::multi_index::ordered_non_unique` Elemente sortiert, bietet die Schnittstelle spezielle Methoden an, um bestimmte Positionen innerhalb der sortierten Elemente zu finden. So werden im Beispiel 12.4 über `lower_bound()` und `upper_bound()` Tiere gesucht, die mindestens vier und höchstens acht Beine haben. Diese Methoden werden von keiner anderen Schnittstelle angeboten, da sie eine Sortierung voraussetzen.
Die dritte Schnittstelle, die im Beispiel 12.4 zur Containerdefinition eingesetzt wird, ist `boost::multi_index::random_access`. Über diese Schnittstelle kann ein MultiIndex-Container wie ein Vektor vom Typ `std::vector` behandelt werden. Die beiden prominentesten Methoden sind `operator[]` und `at()`.
Beachten Sie, dass `boost::multi_index::random_access` die Schnittstelle `boost::multi_index::sequenced` einschließt. Wenn Sie zur Containerdefinition `boost::multi_index::random_access` verwenden, ist es wenig sinnvoll, zusätzlich `boost::multi_index::sequenced` einzusetzen, da sämtliche Methoden dieser Schnittstelle auch über `boost::multi_index::random_access` zur Verfügung stehen.
Nachdem Sie die vier Schnittstellen, die Boost.MultiIndex anbietet, kennengelernt haben, soll im Folgenden ein Blick auf *Schlüsselentnehmer* geworfen werden. Einen Schlüsselentnehmer haben Sie bereits kennengelernt: `boost::multi_index::member` aus der Headerdatei `boost/multi_index/member.hpp`. Diese Hilfsklasse wird Schlüsselentnehmer genannt, da mit ihr angegeben werden kann, welche Eigenschaft einer Klasse als Schlüssel in einer Schnittstelle verwendet werden soll.
Im Beispiel 12.5 lernen Sie zwei weitere Schlüsselentnehmer kennen.

Beispiel 12.5 Die Schlüsselentnehmer `identity` und `const_mem_fun`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/mem_fun.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::multi_index;

class animal
{
public:
  animal(std::string name, int legs) : name_{std::move(name)},
    legs_(legs) {}
  bool operator<(const animal &a) const { return legs_ < a.legs_; }
  const std::string &name() const { return name_; }
private:
  std::string name_;
  int legs_;
};

typedef multi_index_container<
  animal,
  indexed_by<
    ordered_unique<
      identity<animal>
    >,
    hashed_unique<
      const_mem_fun<
        animal, const std::string&, &animal::name
```

```
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.emplace("cat", 4);
  animals.emplace("shark", 0);
  animals.emplace("spider", 8);

  std::cout << animals.begin()->name() << '\n';

  const auto &name_index = animals.get<1>();
  std::cout << name_index.count("shark") << '\n';
}
```

Der Schlüsselentnehmer `boost::multi_index::identity`, definiert in der Headerdatei `boost/multi_index/identity.hpp`, wird verwendet, um den Typ, der im MultiIndex-Container gespeichert wird, selbst als Schlüssel zu verwenden. Für Beispiel 12.5 bedeutet dies, dass die Klasse `animal` sortiert werden können muss, weil der Typ selbst als Schlüssel für die Schnittstelle `boost::multi_index::ordered_unique` dient. Das ist der Grund, warum der Operator `operator<` für die Klasse `animal` definiert ist.
In der Headerdatei `boost/multi_index/mem_fun.hpp` sind die beiden Schlüsselentnehmer `boost::multi_index::const_mem_fun` und `boost::multi_index::mem_fun` definiert, mit denen der Rückgabewert einer Methode als Schlüssel verwendet werden kann. Im Beispiel 12.5 ist dies der Rückgabewert von `name()`. `boost::multi_index::const_mem_fun` wird für konstante Methoden verwendet, `boost::multi_index::mem_fun` für alle anderen Methoden.
Boost.MultiIndex bietet zwei weitere Schlüsselentnehmer namens `boost::multi_index::global_fun` und `boost::multi_index::composite_key` an. Während der erstgenannte Schlüsselentnehmer für freistehende Funktionen und statische Methoden verwendet werden kann, ermöglicht `boost::multi_index::composite_key`, einen Schlüsselentnehmer bestehend aus mehreren anderen Schlüsselentnehmern zu konstruieren.

Kapitel 13

Boost.Bimap

Die Bibliothek Boost.Bimap basiert auf Boost.MultiIndex und bietet einen Container an, den Sie sofort verwenden können. Es handelt sich hierbei um einen Container ähnlich wie `std::map`, wobei nicht einseitig nach Schlüsseln, sondern auch nach Werten gesucht werden kann. Auf die Map kann also von beiden Seiten zugegriffen werden. Was jeweils Schlüssel und Wert ist, hängt davon ab, von welcher Seite der Zugriff erfolgt.

Beispiel 13.1 `boost::bimap` in Aktion

```
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  std::cout << animals.left.count("cat") << '\n';
  std::cout << animals.right.count(8) << '\n';
}
```

Der Container `boost::bimap`, definiert in der Headerdatei `boost/bimap.hpp`, bietet zwei Eigenschaften **`left`** und **`right`** an. Über diese kann auf die beiden Container vom Typ `std::map` zugegriffen werden, die `boost::bimap` in sich vereint. Während im Beispiel 13.1 über **`left`** auf den Container zugegriffen wird, der Strings als Schlüssel verwendet, wird über **`right`** auf den Container zugegriffen, der Zahlen als Schlüssel verwendet.

Neben dem Zugriff auf Elemente sowohl über einen linken als auch rechten Container gestattet `boost::bimap`, Schlüssel/Wert-Paare als Relationen zu betrachten. Sehen Sie sich dazu Beispiel 13.2 an.

Beispiel 13.2 Zugriff auf Relationen

```
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  for (auto it = animals.begin(); it != animals.end(); ++it)
    std::cout << it->left << " has " << it->right << " legs\n";
```

```
}
```

Es ist nicht notwendig, jeweils über **`left`** oder **`right`** auf Elemente im Container zuzugreifen. Sie können auch direkt über Elemente iterieren und über den Iterator auf die linke oder rechte Seite des jeweiligen Elements zugreifen.
Während es für `std::map` einen Container namens `std::multimap` gibt, der mehrere Elemente mit gleichem Schlüssel speichern kann, gibt es keinen entsprechenden Container neben `boost::bimap`. Das bedeutet jedoch nicht, dass es nicht möglich ist, Elemente mit gleichem Schlüssel in einem Container vom Typ `boost::bimap` zu speichern.
Die beiden Template-Parameter, die `boost::bimap` übergeben werden, geben genaugenommen nicht Typen an, die gespeichert werden sollen, sondern Containertypen, die für **`left`** und **`right`** verwendet werden sollen. Wird wie in den bisherigen Beispielen kein Containertyp angegeben, sondern lediglich `std::string` und int, wird automatisch der Containertyp `boost::bimaps::set_of` verwendet. Und dieser erlaubt ähnlich wie `std::map` nur Elemente mit eindeutigem Schlüssel.

Beispiel 13.3 Explizite Angabe von `boost::bimaps::set_of`

```
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>,
    boost::bimaps::set_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  std::cout << animals.left.count("spider") << '\n';
  std::cout << animals.right.count(8) << '\n';
}
```

Im Beispiel 13.3 wird `boost::bimaps::set_of` explizit verwendet.
Neben der Klasse `boost::bimaps::set_of`, die Elemente wie `std::map` sortiert und darauf achtet, dass Schlüssel eindeutig sind, stehen andere Containertypen zur Verfügung, mit denen `boost::bimap` angepasst werden kann.

Beispiel 13.4 Duplikate mit `boost::bimaps::multiset_of` zulassen

```
#include <boost/bimap.hpp>
#include <boost/bimap/multiset_of.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>,
    boost::bimaps::multiset_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"dog", 4});

  std::cout << animals.left.count("dog") << '\n';
  std::cout << animals.right.count(4) << '\n';
}
```

Im Beispiel 13.4 wird der Containertyp `boost::bimaps::multiset_of` verwendet, der in der Headerdatei `boost/bimap/multiset_of.hpp` definiert ist. Er funktioniert ähnlich wie `boost::bimaps::set_of`, nur dass Schlüssel nicht mehr eindeutig sein müssen. Somit gibt Beispiel 13.4 für die Anzahl der vierbeinigen Tiere 2 aus.

Beachten Sie, dass Sie für andere Containertypen als `boost::bimaps::set_of` die entsprechenden Headerdateien einbinden müssen. Weil `boost::bimaps::set_of` standardmäßig in Containern vom Typ `boost::bimap` verwendet wird, müssen Sie die entsprechende Headerdatei `boost/bimap/set_of.hpp` nicht einbinden.
Neben den bisher kennengelernten Containertypen bietet Boost.Bimap die Klassen `boost::bimaps::unordered_set_of`, `boost::bimaps::unordered_multiset_of`, `boost::bimaps::list_of`, `boost::bimaps::vector_of` und `boost::bimaps::unconstrainted_set_of` an. Bis auf die Klasse `boost::bimaps::unconstrainted_set_of`, die weiterer Erläuterungen bedarf, funktionieren alle genannten Containertypen wie die aus der Standardbibliothek bekannten Klassen.

Beispiel 13.5 Eine Seite mit `boost::bimaps::unconstrainted_set_of` deaktivieren

```
#include <boost/bimap.hpp>
#include <boost/bimap/unconstrained_set_of.hpp>
#include <boost/bimap/support/lambda.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string,
    boost::bimaps::unconstrained_set_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  auto it = animals.left.find("cat");
  animals.left.modify_key(it, boost::bimaps::_key = "dog");

  std::cout << it->first << '\n';
}
```

Mit `boost::bimaps::unconstrainted_set_of` kann wie im Beispiel 13.5 eine Seite von `boost::bimap` deaktiviert werden. Es ist dann nicht mehr möglich, über **`right`** auf die rechte Seite des Containers zuzugreifen und Tiere nach der Anzahl ihrer Beine zu durchsuchen. In diesem Fall verhält sich der Container vom Typ `boost::bimap` wie `std::map`.
Warum es trotzdem sinnvoll sein kann, statt `std::map` `boost::bimap` zu verwenden, sehen Sie anhand des Beispiels. Da Boost.Bimap auf Boost.MultiIndex basiert, stehen aus Boost.MultiIndex bekannte Methoden zur Verfügung. So wird im Beispiel 13.5 über `modify_key()` ein Schlüssel geändert – etwas, was mit `std::map` nicht möglich ist.
Beachten Sie außerdem, wie der Schlüssel im Detail geändert wird: Über **`boost::bimaps::_key`**, das in der Headerdatei `boost/bimap/support/lambda.hpp` definiert ist, wird auf den aktuellen Schlüssel zugegriffen und ihm ein neuer Wert zugewiesen. Es handelt sich hierbei um eine Lambda-Funktion, die ohne C++11 und andere Boost-Bibliotheken auskommt.
Neben **`boost::bimaps::_key`** ist in der Headerdatei `boost/bimap/support/lambda.hpp` auch **`boost::bimaps::_data`** definiert. Beim Aufruf der Methode `modify_data()` können Sie entsprechend einen Wert in einem Container vom Typ `boost::bimap` ändern.

Kapitel 14

Boost.Array

Die Bibliothek Boost.Array stellt in der Headerdatei `boost/array.hpp` eine Klasse `boost::array` zur Verfügung. `boost::array` ist identisch zur Klasse `std::array`, die mit C++11 in die Standardbibliothek aufgenommen wurde. Sie können `boost::array` ignorieren, wenn Sie in einer C++11-Entwicklungsumgebung arbeiten.
Mit `boost::array` ist es möglich, ein Array zu erstellen, das die gleichen Eigenschaften besitzt wie ein herkömmliches C-Array. Weil `boost::array` jedoch Anforderungen erfüllt, wie sie an im C++-Standard definierte Container gestellt werden, wird der Umgang mit einem Array ähnlich einfach wie mit jedem anderen Container. Grundsätzlich können Sie sich `boost::array` wie `std::vector` vorstellen mit dem entscheidenden Unterschied, dass die Anzahl der Elemente in `boost::array` konstant ist.

Beispiel 14.1 Verschiedene Methoden von `boost::array`

```
#include <boost/array.hpp>
#include <string>
#include <algorithm>
#include <iostream>

int main()
{
  typedef boost::array<std::string, 3> array;
  array a;

  a[0] = "cat";
  a.at(1) = "shark";
  *a.rbegin() = "spider";

  std::sort(a.begin(), a.end());

  for (const std::string &s : a)
    std::cout << s << '\n';

  std::cout << a.size() << '\n';
  std::cout << a.max_size() << '\n';
}
```

Beispiel 14.1 bedarf keiner Erklärungen, da die aufgerufenen Methoden die gleiche Bedeutung haben wie bei `std::vector`.

Kapitel 15

Boost.Unordered

Boost.Unordered stellt die Klassen `boost::unordered_set`, `boost::unordered_multiset`, `boost::unordered_map` und `boost::unordered_multimap` zur Verfügung. Die Klassen sind identisch mit den Hash-Containern, die mit C++11 in die Standardbibliothek aufgenommen wurden. Sie können die Container aus Boost.Unordered ignorieren, wenn Sie in einer C++11-Entwicklungsumgebung arbeiten.

Beispiel 15.1 `boost::unordered_set` in Aktion

```
#include <boost/unordered_set.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::unordered_set<std::string> unordered_set;
  unordered_set set;

  set.emplace("cat");
  set.emplace("shark");
  set.emplace("spider");

  for (const std::string &s : set)
    std::cout << s << '\n';

  std::cout << set.size() << '\n';
  std::cout << set.max_size() << '\n';

  std::cout << std::boolalpha << (set.find("cat") != set.end()) << '\n';
  std::cout << set.count("shark") << '\n';
}
```

`boost::unordered_set` könnte im Beispiel 15.1 durch `std::unordered_set` ersetzt werden. `boost::unordered_set` unterscheidet sich nicht von `std::unordered_set`.

Beispiel 15.2 `boost::unordered_map` in Aktion

```
#include <boost/unordered_map.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::unordered_map<std::string, int> unordered_map;
  unordered_map map;

  map.emplace("cat", 4);
  map.emplace("shark", 0);
  map.emplace("spider", 8);

  for (const auto &p : map)
    std::cout << p.first << ";" << p.second << '\n';
```

```
  std::cout << map.size() << '\n';
  std::cout << map.max_size() << '\n';

  std::cout << std::boolalpha << (map.find("cat") != map.end()) << '\n';
  std::cout << map.count("shark") << '\n';
}
```

Im Beispiel 15.2 wird `boost::unordered_map` verwendet, um zusätzlich zum Namen die Anzahl der Beine jedes Tiers zu speichern. Auch in diesem Fall könnte `boost::unordered_map` durch `std::unordered_map` ersetzt werden.

Beispiel 15.3 Benutzerdefinierter Typ mit Boost.Unordered

```
#include <boost/unordered_set.hpp>
#include <string>
#include <cstddef>

struct animal
{
  std::string name;
  int legs;
};

bool operator==(const animal &lhs, const animal &rhs)
{
  return lhs.name == rhs.name && lhs.legs == rhs.legs;
}

std::size_t hash_value(const animal &a)
{
  std::size_t seed = 0;
  boost::hash_combine(seed, a.name);
  boost::hash_combine(seed, a.legs);
  return seed;
}

int main()
{
  typedef boost::unordered_set<animal> unordered_set;
  unordered_set animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});
}
```

Im Beispiel 15.3 sollen Elemente vom Typ `animal` in einem Container vom Typ `boost::unordered_set` gespeichert werden. Weil die Hashfunktion von `boost::unordered_set` die Klasse `animal` nicht kennt, können keine Hashwerte für Elemente dieses Typs automatisch errechnet werden. Deswegen muss eine Hashfunktion definiert werden – andernfalls würde der Code nicht kompilieren.
Der Name der Hashfunktion, die definiert werden muss, lautet `hash_value()`. Sie muss als einzigen Parameter ein Objekt des Typs erwarten, von dem ein Hashwert gebildet werden soll. Der Typ des Rückgabewerts von `hash_value()` muss std::size_t sein.
Die Funktion `hash_value()` wird automatisch aufgerufen, wenn der Hashwert für ein Objekt errechnet werden soll. Diese Funktion ist für verschiedene Typen in den Boost-Bibliotheken definiert – unter anderem für `std::string`. Für benutzerdefinierte Typen wie `animal` muss sie vom Entwickler definiert werden.
Die Definition von `hash_value()` ist üblicherweise sehr einfach: Der Hashwert wird gebildet, indem nacheinander auf die verschiedenen Bestandteile des Objekts zugegriffen wird. Dies geschieht über die Funktion `boost::hash_combine()`, die aus der Bibliothek Boost.Hash stammt und in der Headerdatei `boost/functional/hash.hpp` definiert ist. Sie müssen diese Headerdatei nicht einbinden, wenn Sie Boost.Unordered verwenden, da sämtliche Container in dieser Bibliothek zum Errechnen von Hashwerten auf Boost.Hash zugreifen.

Neben der Definition der Funktion `hash_value()` muss es außerdem möglich sein, zwei Objekte mit `==` zu vergleichen. Deswegen ist für die Klasse `animal` im Beispiel 15.3 der Operator `operator==` überladen.
Die Hash-Container der C++11-Standardbibliothek verwenden eine Hashfunktion aus der Headerdatei `functional`. Die Hash-Container aus Boost.Unordered erwarten die Hashfunktion `hash_value()`. Ob Sie innerhalb von `hash_value()` auf Boost.Hash zugreifen, spielt keine Rolle. Boost.Hash bietet sich an, da Funktionen wie `boost::hash_combine()` es einfacher machen, einen Hashwert schrittweise basierend auf mehreren Eigenschaften zu errechnen. Hierbei handelt es sich jedoch um ein Implementationsdetail von `hash_value()`. Abgesehen von den unterschiedlichen Hashfunktionen, die erwartet werden, sind die Hash-Container in Boost.Unordered und der Standardbibliothek grundsätzlich gleich.

Kapitel 16

Boost.CircularBuffer

Die Bibliothek Boost.CircularBuffer bietet einen *Ringspeicher* an. Dabei handelt es sich um einen Container mit folgenden zwei wesentlichen Eigenschaften:

- Die Kapazität des Ringspeichers ist konstant und wird von Ihnen vorgegeben. Die Kapazität ändert sich nicht automatisch, wenn Sie eine Methode wie `push_back()` aufrufen. Nur Sie können die Kapazität ändern. Die Größe des Ringspeichers wird durch die Kapazität begrenzt.
- Trotz konstanter Kapazität kann beliebig oft `push_back()` aufgerufen werden, um Elemente im Ringspeicher abzulegen. Ist die maximale Größe erreicht und der Ringspeicher voll, werden Elemente überschrieben.

Ein Ringspeicher bietet sich an, wenn der vorhandene Speicherplatz begrenzt ist und verhindert werden soll, dass ein Container beliebig groß wird. Auch dann, wenn alte Daten keine Rolle mehr spielen, weil zwischenzeitlich genügend neue Daten vorliegen, kann ein Ringspeicher eine gute Wahl sein. So wird Speicherplatz automatisch wiederverwendet, und alte Daten werden von neuen überschrieben.
Um den Ringspeicher der Bibliothek Boost.CircularBuffer nutzen zu können, muss die Headerdatei `boost/circular_buffer.hpp` eingebunden werden. Die Headerdatei stellt die Klasse `boost::circular_buffer` zur Verfügung.

Beispiel 16.1 `boost::circular_buffer` in Aktion

```
#include <boost/circular_buffer.hpp>
#include <iostream>

int main()
{
  typedef boost::circular_buffer<int> circular_buffer;
  circular_buffer cb{3};

  std::cout << cb.capacity() << '\n';
  std::cout << cb.size() << '\n';

  cb.push_back(0);
  cb.push_back(1);
  cb.push_back(2);

  std::cout << cb.size() << '\n';

  cb.push_back(3);
  cb.push_back(4);
  cb.push_back(5);

  std::cout << cb.size() << '\n';

  for (int i : cb)
    std::cout << i << '\n';
}
```

`boost::circular_buffer` ist ein Template, das den Typ der zu speichernden Elemente als Parameter erwartet. So können im Beispiel 16.1 im Ringspeicher **`cb`** Ganzzahlen vom Typ int abgelegt werden.
Die Kapazität des Ringspeichers wird nicht als Template-Parameter angegeben. Stattdessen legen Sie die Kapazität bei der Instanziierung des Ringspeichers fest. Während der Standardkonstruktor von `boost::circular_buffer` einen Ringspeicher mit einer Kapazität von null Elementen erstellt, steht ein Konstruktor zur Verfügung, um die Kapazität auf einen beliebigen Wert zu setzen. Dieser Konstruktor wird im Beispiel 16.1 verwendet: Der Ringspeicher **`cb`** erhält eine Kapazität von drei Elementen.
Sie erhalten die Kapazität eines Ringspeichers, indem Sie die Methode `capacity()` aufrufen. Im Beispiel 16.1 gibt `capacity()` folglich 3 zurück.
Die Kapazität ist nicht gleichbedeutend mit der Anzahl gespeicherter Elemente. Während der Rückgabewert von `capacity()` konstant ist, kann `size()` unterschiedliche Werte zurückgeben. Der Rückgabewert von `size()` liegt immer zwischen einschließlich 0 und der Kapazität des Ringspeichers.
Wenn Sie Beispiel 16.1 ausführen, sehen Sie, dass der erste Aufruf von `size()` 0 zurückgibt. Zu diesem Zeitpunkt sind keine Elemente im Ringspeicher gespeichert. Nachdem dreimal `push_back()` aufgerufen wurde, befinden sich drei Elemente im Ringspeicher – der zweite Aufruf von `size()` gibt 3 zurück. Der wiederholte Aufruf von `push_back()` führt nicht dazu, dass der Speicher wächst. Die drei neuen Zahlen überschreiben die bisher gespeicherten drei Zahlen. Deswegen gibt `size()` auch beim dritten Aufruf 3 zurück.
Zur Kontrolle werden am Ende des Programms alle gespeicherten Zahlen ausgegeben. Dabei werden lediglich die Zahlen 3, 4 und 5 in die Standardausgabe geschrieben. Die zuerst gespeicherten Zahlen 0, 1 und 2 wurden von diesen überschrieben.

Beispiel 16.2 Verschiedene Methoden von `boost::circular_buffer`

```
#include <boost/circular_buffer.hpp>
#include <iostream>

int main()
{
  typedef boost::circular_buffer<int> circular_buffer;
  circular_buffer cb{3};

  cb.push_back(0);
  cb.push_back(1);
  cb.push_back(2);
  cb.push_back(3);

  std::cout << std::boolalpha << cb.is_linearized() << '\n';

  circular_buffer::array_range ar1, ar2;

  ar1 = cb.array_one();
  ar2 = cb.array_two();
  std::cout << ar1.second << ";" << ar2.second << '\n';

  for (int i : cb)
    std::cout << i << '\n';

  cb.linearize();

  ar1 = cb.array_one();
  ar2 = cb.array_two();
  std::cout << ar1.second << ";" << ar2.second << '\n';
}
```

Im Beispiel 16.2 werden einige Methoden eingesetzt, die nicht von anderen Containern bekannt sind. Es handelt sich hierbei um die Methoden `is_linearized()`, `array_one()`, `array_two()` und `linearize()`. Diese Methoden sind deswegen interessant, weil sie die Funktionsweise des Ringspeichers verdeutlichen.
Ein Ringspeicher ähnelt `std::vector`. Die Kapazität ist zwar konstant, und Elemente werden überschrieben, wenn der Speicher voll ist. Die Implementation von `boost::circular_buffer` könnte dennoch problemlos auf `std::vector` basieren. Während der Anfang und das Ende eines Vektors jedoch klar definiert sind, ist dies beim Ringspeicher nicht der Fall. So können Sie einen Vektor wie ein herkömmliches C-Array behandeln, da alle Elemente in einem zusammenhängenden Speicherblock stehen und sich das erste Element an der kleinsten Speicheradresse und das letzte Element an der größten Speicheradresse befindet. Dies ist beim Ringspeicher nicht

garantiert.
Bei einem Ringspeicher von Anfang und Ende zu sprechen, mag sich merkwürdig anhören. Da Sie wie bereits gesehen bei einem Ringspeicher über einen Iterator auf Elemente zugreifen können – `boost::circular_buffer` bietet entsprechende Methoden wie `begin()` und `end()` an – hat auch ein Ringspeicher einen Anfang und ein Ende. Während Sie sich bei Iteratoren keine Gedanken machen müssen, wo sich der Anfang und das Ende befinden – bei Iteratoren funktioniert alles wie gewohnt automatisch – müssen Sie sich bei einem Zugriff per Zeiger auf Elemente im Ringspeicher etwas den Kopf zerbrechen. Oder Sie verwenden die vier genannten Methoden `is_linearized()`, `array_one()`, `array_two()` und `linearize()`.
Die Methode `is_linearized()` gibt `true` zurück, wenn der Anfang des Ringspeichers an der kleinsten Speicheradresse liegt. In diesem Fall befinden sich sämtliche Elemente im Ringspeicher von Anfang bis Ende nacheinander an aufsteigenden Speicheradressen. Sie können demnach auf Elemente im Ringspeicher wie bei einem herkömmlichen C-Array zugreifen.
Gibt `is_linearized()` `false` zurück, liegt der Anfang des Ringspeichers nicht an der kleinsten Speicheradresse. Dies ist im Beispiel 16.2 der Fall. Während die ersten drei Elemente 0, 1 und 2 in genau dieser Reihenfolge im Speicher liegen, führt der vierte Aufruf von `push_back()` dazu, dass die Zahl 3 die im Ringspeicher abgelegte Zahl 0 überschreibt. Da 3 die letzte mit `push_back()` hinzugefügte Zahl ist, handelt es sich hierbei um das neue Ende des Ringspeichers. Den Anfang markiert die Zahl 1, die an der nächst höheren Speicheradresse liegt. Die Daten sind demnach nicht mehr nacheinander an aufsteigenden Speicheradressen gespeichert.
Wenn das Ende des Ringspeichers an einer niedrigeren Speicheradresse als der Anfang des Ringspeichers liegt, kann auf die Elemente nicht mehr wie bei einem herkömmlichen C-Array zugegriffen werden. Stattdessen muss auf zwei C-Arrays zugegriffen werden. Um die Positionen und Größen dieser Arrays nicht selbst berechnen zu müssen, stellt `boost::circular_buffer` die beiden Methoden `array_one()` und `array_two()` zur Verfügung.
`array_one()` und `array_two()` geben ein `std::pair` zurück, dessen erstes Element ein Zeiger auf das Array ist und dessen zweites Element die Größe des jeweiligen Arrays angibt. Über `array_one()` kann auf das Array zugegriffen werden, das an der kleinsten Adresse im Ringspeicher beginnt – über `array_two()` entsprechend auf das Array, das an der größten Adresse im Ringspeicher endet.
Ist der Ringspeicher linearisiert – gibt also `is_linearized()` `true` zurück – darf zwar auch `array_two()` aufgerufen werden. Der Aufruf ist jedoch wenig sinnvoll, da es nur ein Array im Ringspeicher gibt. Das zweite Array enthält demnach null Elemente.
Wenn Sie einen Ringspeicher wie ein herkömmliches C-Array behandeln möchten, können Sie eine Neuordnung erzwingen, indem Sie `linearize()` aufrufen. Dann ist garantiert, dass Sie über `array_one()` auf alle im Ringspeicher abgelegten Elemente zugreifen können und nicht zusätzlich `array_two()` aufrufen müssen.
Boost.CircularBuffer bietet mit `boost::circular_buffer_space_optimized` eine weitere Klasse an, um einen Ringspeicher zu erstellen. Diese Klasse wird genauso verwendet wie `boost::circular_buffer` und ist ebenfalls in der Headerdatei `boost/circular_buffer.hpp` definiert. `boost::circular_buffer_space_optimized` reserviert jedoch keinen Speicher bei der Instanziierung. Stattdessen wächst der Speicher dynamisch, bis die Kapazitätsgrenze erreicht ist. Werden Elemente gelöscht, schrumpft der Speicher entsprechend. `boost::circular_buffer_space_optimized` geht demnach effizienter mit Speicher um und kann eine gute Wahl sein, wenn ein Ringspeicher zum Beispiel eine sehr große Kapazität hat, die von einem Programm unter Umständen nicht benötigt wird.

Kapitel 17

Boost.Heap

Boost.Heap hätte auch Boost.PriorityQueue genannt werden können: Die Bibliothek stellt mehrere Priority-Queues zur Verfügung. Diese unterscheiden sich von `std::priority_queue` dahingehend, dass sie mehr Funktionen unterstützen.

Beispiel 17.1 `boost::heap::priority_queue` in Aktion

```
#include <boost/heap/priority_queue.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  priority_queue<int> pq;
  pq.push(2);
  pq.push(3);
  pq.push(1);

  for (int i : pq)
    std::cout << i << '\n';

  priority_queue<int> pq2;
  pq2.push(4);
  std::cout << std::boolalpha << (pq > pq2) << '\n';
}
```

Beispiel 17.1 verwendet die Klasse `boost::heap::priority_queue`, die in der Headerdatei `boost/heap/priority_queue.hpp` definiert ist. Diese verhält sich grundsätzlich wie `std::priority_queue`, ermöglicht es aber, über die Elemente in der Queue zu iterieren. Die Reihenfolge der Elemente in der Iteration ist dabei zufällig.

Objekte vom Typ `boost::heap::priority_queue` können außerdem miteinander verglichen werden. Der Vergleich im Beispiel 17.1 gibt `true` zurück, weil **`pq`** mehr Elemente als **`pq2`** hat. Hätten beide Queues gleich viele Elemente, würden diese paarweise miteinander verglichen werden.

Beispiel 17.2 `boost::heap::binomial_heap` in Aktion

```
#include <boost/heap/binomial_heap.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  binomial_heap<int> bh;
  bh.push(2);
  bh.push(3);
  bh.push(1);

  binomial_heap<int> bh2;
  bh2.push(4);
```

```
  bh.merge(bh2);

  for (auto it = bh.ordered_begin(); it != bh.ordered_end(); ++it)
    std::cout << *it << '\n';
  std::cout << std::boolalpha << bh2.empty() << '\n';
}
```

Beispiel 17.2 stellt die Klasse `boost::heap::binomial_heap` vor. Diese Implementation einer Priority-Queue ermöglicht es nicht nur, geordnet über Elemente zu iterieren – also von Elementen mit der höchsten Priorität zu Elementen mit der niedrigsten. Priority-Queues vom Typ `boost::heap::binomial_heap` können verschmolzen werden. Dabei werden die Elemente der einen Queue zur anderen Queue hinzugefügt.
Im Beispiel wird für die Queue **bh** `merge()` aufgerufen und die Queue **bh2** als Parameter übergeben. Der Aufruf von `merge()` führt dazu, dass die Zahl 4 von **bh2** nach **bh** verschoben wird. **bh** enthält nach dem Aufruf vier Zahlen, während **bh2** leer ist.
In der `for`-Schleife wird mit `ordered_begin()` und `ordered_end()` auf **bh** zugegriffen. `ordered_begin()` gibt einen Iterator zurück, mit dem über Elemente von höchster nach niedrigster Priorität iteriert werden kann. Beispiel 17.2 gibt demnach in der Schleife die Zahlen 4, 3, 2 und 1 in dieser Reihenfolge aus.

Beispiel 17.3 Elemente in einer `boost::heap::binomial_heap` ändern

```
#include <boost/heap/binomial_heap.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  binomial_heap<int> bh;
  auto handle = bh.push(2);
  bh.push(3);
  bh.push(1);

  bh.update(handle, 4);

  std::cout << bh.top() << '\n';
}
```

`boost::heap::binomial_heap` ermöglicht es, Elemente zu ändern, nachdem sie der Queue hinzugefügt wurden. So wird im Beispiel 17.3 ein Handle gespeichert, der von `push()` zurückgegeben wird. Dieser Handle ermöglicht es, auf die in **bh** gespeicherte Zahl 2 zuzugreifen.
Mit `update()` bietet `boost::heap::binomial_heap` eine Methode an, der ein Handle übergeben werden kann, um das entsprechende Element zu ändern. Im Beispiel wird die Methode aufgerufen, um die Zahl 2 durch die Zahl 4 zu ersetzen. Wird anschließend mit `top()` das Element mit der höchsten Priorität abgefragt, wird 4 zurückgegeben.
`boost::heap::binomial_heap` bietet neben `update()` weitere Methoden an, um Elemente zu ändern. So können anstatt `update()` die Methoden `increase()` oder `decrease()` aufgerufen werden, wenn der Aufrufer weiß, dass die Änderung zu einer höheren oder niedrigeren Priorität führt. Im Beispiel 17.3 könnte `update()` durch `increase()` ersetzt werden, weil die Zahl von 2 auf 4 erhöht wird.
Neben `boost::heap::priority_queue` und `boost::heap::binomial_heap` stellt Boost.Heap weitere Implementationen zur Verfügung. Diese unterscheiden sich größtenteils in der Laufzeitkomplexität der verschiedenen Funktionen. So können Sie zum Beispiel eine Klasse `boost::heap::fibonacci_heap` verwenden, wenn Sie eine konstante Laufzeitkomplexität von `push()` wünschen. Die Dokumentation von Boost.Heap stellt die Laufzeitkomplexitäten der verschiedenen Klassen und Funktionen übersichtlich in einer Tabelle gegenüber.

Kapitel 18

Boost.Intrusive

Bei Boost.Instrusive handelt es sich um eine Bibliothek, die sich besonders für den Einsatz in Programmen eignet, die eine hohe Performance erzielen müssen. Die Bibliothek stellt Werkzeuge zur Verfügung, um *intrusive Container* zu erstellen. Diese Container ersetzen die bekannten Container der Standardbibliothek. Sie haben den Nachteil, dass sie nicht ganz so einfach einzusetzen sind wie beispielsweise `std::list` oder `std::set`. Dafür bieten sie diese Vorteile:

- Intrusive Container reservieren keinen dynamischen Speicher. Ein Aufruf von `push_back()` führt nicht zu einer dynamischen Speicherallokation mit `new`. Das ist ein Grund, warum der Einsatz intrusiver Container zu einer höheren Performance führen kann.
- Intrusive Container speichern keine Kopien von Objekten, sondern Originalobjekte. Dies ergibt sich bereits daraus, dass keine Speicherallokation stattfindet. Daraus resultiert aber ein weiterer Vorteil: Methoden wie `push_back()` werfen keine Ausnahmen. So kann es weder zu einer fehlgeschlagenen Speicherallokation kommen noch zu Problemen beim Kopieren von Objekten – weder das eine noch das andere geschieht bei intrusiven Containern.

Die Vorteile werden durch einen etwas komplizierteren Code erkauft, der die notwendigen Vorbedingungen schafft, damit Objekte bestimmter Typen in intrusiven Containern gespeichert werden können. So ist es nicht möglich, Objekte beliebiger Typen in intrusiven Containern zu speichern. Sie können zum Beispiel keine Strings vom Typ `std::string` in einem intrusiven Container ablegen. Hier müssen Sie auf die bekannten Container aus der Standardbibliothek zugreifen.
Sehen Sie sich Beispiel 18.1 an, in dem eine Klasse `animal` so präpariert wird, dass Objekte dieses Typs in einer intrusiven Liste gespeichert werden können.

Beispiel 18.1 `boost::intrusive::list` in Aktion

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal a3{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;
```

```
  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(a3);

  a1.name = "dog";

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Eine Liste ist so definiert, dass man von einem Element der Liste auf das nächste zugreifen kann. Dies wird üblicherweise mit einem Zeiger implementiert. Wenn eine intrusive Liste Objekte vom Typ `animal` speichern können soll, ohne dynamisch Speicher zu reservieren, muss es irgendwo einen Zeiger geben, um sich später in der Liste von einem Element zum nächsten zu hangeln.
Um Objekte vom Typ `animal` in einer intrusiven Liste speichern zu können, muss die Klasse selbst Variablen bereitstellen, die es der intrusiven Liste ermöglichen, Elemente zu verketten. Boost.Intrusive stellt hierzu *Hooks* zur Verfügung – Klassen, von denen abgeleitet werden kann, um die notwendigen Variablen zu erben und sie so bereitzustellen. So muss `animal` von der Klasse `boost::intrusive::list_base_hook` abgeleitet werden, damit Objekte vom Typ `animal` in einer intrusiven Liste verwaltet werden können. Dank dieser Vorgehensweise ist es nicht notwendig zu wissen, welche Details die intrusive Liste im Einzelnen benötigt. Es kann aber davon ausgegangen werden, dass `boost::intrusive::list_base_hook` mindestens zwei Zeiger zur Verfügung stellt, da `boost::intrusive::list` eine doppelt verkettete Liste ist. Dank der Elternklasse `boost::intrusive::list_base_hook` stellt `animal` diese beiden Zeiger zur Verfügung, die notwendig sind, um Elemente dieses Typs zu verketten.
Beachten Sie, dass es sich bei `boost::intrusive::list_base_hook` um ein Template handelt. Da das Template Standardwerte besitzt, müssen Sie keine Werte bei der Instanziierung angeben.
Boost.Intrusive stellt die Klasse `boost::intrusive::list` zur Verfügung, um eine intrusive Liste zu erstellen. Die Klasse ist in der Headerdatei `boost/intrusive/list.hpp` definiert und wird grundsätzlich genauso verwendet wie `std::list`. So können Elemente nicht nur mit `push_back()` der Liste hinzugefügt werden. Es kann auch über Elemente in der Liste iteriert werden.
Es ist wichtig zu verstehen, dass intrusive Container keine Kopien, sondern Originalobjekte speichern. So gibt Beispiel 18.1 die Namen `dog`, `shark` und `spider` aus – nicht `cat`. Es ist das Objekt **`a1`** selbst, das in die Liste eingebunden ist. Deswegen ist die Namensänderung sichtbar, wenn über die Elemente in der Liste iteriert wird und die Namen ausgegeben werden.
Da intrusive Container keine Kopien speichern, müssen Sie darauf achten, dass Sie Objekte explizit aus einem intrusiven Container entfernen, wenn Sie sie zerstören.

Beispiel 18.2 Entfernen und Zerstören dynamisch reservierter Objekte

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;
```

```
  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  animals.pop_back();
  delete a3;

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Im Beispiel 18.2 wird ein Objekt vom Typ `animal` mit `new` erstellt und der Liste **`animals`** hinzugefügt. Wenn Sie dieses Objekt mit `delete` zerstören wollen, weil es nicht mehr benötigt wird, müssen Sie es aus der Liste entfernen. Achten Sie darauf, dass Sie das Objekt zuerst aus der Liste entfernen, bevor Sie es zerstören – die Reihenfolge ist wichtig. Andernfalls laufen Sie Gefahr, dass im intrusiven Container über Zeiger auf ungültige Speicherbereiche zugegriffen wird, in denen sich kein Objekt vom Typ `animal` mehr befindet.
Da intrusive Container Speicher weder reservieren noch freigeben, gilt natürlich auch, dass Objekte, die in einem intrusiven Container verwaltet werden, weiter existieren, wenn der intrusive Container zerstört wird.
Weil bei intrusiven Containern das Entfernen von Elementen nicht automatisch mit dem Zerstören dieser verbunden ist, werden neben den von Containern aus der Standardbibliothek bekannten Methoden weitere angeboten. Zu diesen gehört zum Beispiel `pop_back_and_dispose()`.

Beispiel 18.3 Entfernen und Zerstören mit `pop_back_and_dispose()`

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  animals.pop_back_and_dispose([](animal *a){ delete a; });

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

`pop_back_and_dispose()` entfernt ein Element aus der Liste und zerstört es. Da intrusive Container nicht wissen können, wie ein Element der Liste zerstört werden muss, muss eine Funktion oder ein Funktionsobjekt als Parameter an `pop_back_and_dispose()` übergeben werden. Die Methode ruft die Funktion oder das Funktionsobjekt auf und übergibt einen Zeiger auf das Objekt, das nun nicht mehr Teil der Liste ist und zerstört werden kann. Im Beispiel 18.3 wird eine Lambda-Funktion übergeben, die lediglich `delete` ausführt.

Beachten Sie, dass im Beispiel 18.3 ausschließlich das dritte Element in **`animals`** wie gezeigt mit `pop_back_and_dispose()` entfernt werden darf. Die anderen beiden Elemente in der Liste sind nicht mit `new` erstellt worden und dürfen demnach nicht mit `delete` zerstört werden.
Boost.Intrusive bietet eine weitere Möglichkeit an, das Entfernen und Zerstören von Elementen zu verbinden.

Beispiel 18.4 Entfernen und Zerstören mit Auto-Unlink-Modus

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

typedef link_mode<auto_unlink> mode;

struct animal : public list_base_hook<mode>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef constant_time_size<false> constant_time_size;
  typedef list<animal, constant_time_size> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  delete a3;

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Hook-Klassen kann ein Parameter für einen Link-Modus übergeben werden. Dazu wird auf die Klasse `boost::intrusive::link_mode` zugegriffen, die ihrerseits ein Template ist und einen Parameter erwartet. Indem Sie `boost::intrusive::auto_unlink` übergeben, wird der Auto-Unlink-Modus aktiviert.
Im Auto-Unlink-Modus wird ein Element aus einem intrusiven Container automatisch entfernt, wenn es zerstört wird. So gibt Beispiel 18.4 nur die Namen `cat` und `shark` aus.
Der Auto-Unlink-Modus kann nur verwendet werden, wenn die Methode `size()`, die von allen intrusiven Containern angeboten wird, keine konstante Komplexität besitzt. Standardmäßig besitzt diese Methode eine konstante Komplexität, was bedeutet: Egal, wie viele Elemente sich in einem Container befinden – die Anzahl der Elemente wird von `size()` immer gleich schnell berechnet. Hierbei handelt es sich um eine weitere Einstellungsmöglichkeit, um die Performance zu optimieren.
Um die Komplexität der Methode `size()` zu ändern, muss auf die Klasse `boost::intrusive::constant_time_size` zugegriffen werden. Es handelt sich bei dieser Klasse um ein Template, das lediglich `true` oder `false` als Parameter erwartet. Die Klasse `boost::intrusive::constant_time_size` kann als zweiter Template-Parameter an intrusive Container wie `boost::intrusive::list` übergeben werden, um die Komplexität der Methode `size()` einzustellen.
Sie haben soeben zwei neue Konzepte kennengelernt: Intrusive Container besitzen Link-Modi und eine Einstellungsmöglichkeit zur Komplexität von `size()`. Auch wenn dies den Anschein erweckt, als würde Boost.Intrusive von Konfigurationseinstellungen nur so wimmeln, gibt es nicht viel mehr zu entdecken. Zum Beispiel gibt es insgesamt nur drei Link-Modi, von denen Sie lediglich den Auto-Unlink-Modus explizit kennen müssen – der Standard-Modus, der immer dann verwendet wird, wenn Sie keinen Link-Modus vorgeben, ist in allen anderen

Fällen eine gute Wahl.
Des Weiteren gibt es keine Einstellungsmöglichkeiten bezüglich anderer Methoden, die von intrusiven Containern angeboten werden. Es gibt also keine weiteren Klassen neben `boost::intrusive::constant_time_size`, die Sie kennenlernen müssen.
Im Beispiel 18.5 soll nicht nur ein anderer Hook-Mechanismus vorgestellt werden. Mit `boost::intrusive::set` wird außerdem ein anderer intrusiver Container verwendet.

Beispiel 18.5 Ein Hook für `boost::intrusive::set` als Eigenschaft definiert

```
#include <boost/intrusive/set.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal
{
  std::string name;
  int legs;
  set_member_hook<> set_hook;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal a3{"spider", 8};

  typedef member_hook<animal, set_member_hook<>, &animal::set_hook> hook;
  typedef set<animal, hook> animal_set;
  animal_set animals;

  animals.insert(a1);
  animals.insert(a2);
  animals.insert(a3);

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Boost.Intrusive bietet grundsätzlich zwei Möglichkeiten an, eine Klasse mit einem Hook zu versehen: Entweder leiten Sie von einer Hook-Klasse ab oder Sie verwenden eine Hook-Klasse, um eine Eigenschaft zu definieren. Während in den vorherigen Beispielen von `boost::intrusive::list_base_hook` abgeleitet wurde, verwendet Beispiel 18.5 die Klasse `boost::intrusive::set_member_hook`, um eine Eigenschaft zu definieren. Beachten Sie, dass der Name der Eigenschaft keine Rolle spielt. Sie müssen jedoch je nach intrusivem Container eine andere Hook-Klasse verwenden. So heißt zum Beispiel die Hook-Klasse, um eine Eigenschaft für eine intrusive Liste zu definieren, nicht `boost::intrusive::set_member_hook`, sondern `boost::intrusive::list_member_hook`.
Intrusive Container haben unterschiedliche Hook-Klassen, weil Container unterschiedliche Anforderungen an Elemente stellen, die sie speichern sollen. Sie können jedoch problemlos mehrere Hook-Klassen verwenden, um Objekte in unterschiedlichen intrusiven Containern speichern zu können. Mit `boost::intrusive::any_base_hook` und `boost::intrusive::any_member_hook` stehen sogar Klassen zur Verfügung, um Objekte in jedem beliebigen intrusiven Container speichern zu können, ohne dass von mehreren Hook-Klassen abgeleitet werden muss oder mehrere Eigenschaften als Hooks definiert werden müssen.
Intrusive Container erwarten standardmäßig, dass Hooks in Elternklassen definiert sind. Verwenden Sie wie im Beispiel 18.5 eine Eigenschaft, um einen Hook zu definieren, müssen Sie dem intrusiven Container mitteilen, welche Eigenschaft als Hook verwendet werden soll. Deswegen wird dem Template `boost::intrusive::set` nicht nur die Klasse `animal` als Parameter übergeben, sondern auch der Typ hook. Dieser ist mit Hilfe der Klasse `boost::intrusive::member_hook` definiert, die zum Einsatz kommt, wenn ein intrusiver Container eine Eigenschaft als Hook verwenden soll. Das Template `boost::intrusive::member_hook` erwartet dabei den

Klassennamen der im Container zu speichernden Elemente, den Typ des Hooks und einen Zeiger auf die entsprechende Eigenschaft als Parameter.
Wenn Sie Beispiel 18.5 ausführen, wird `shark`, `cat` und `spider` in dieser Reihenfolge ausgegeben.
Neben den in diesem Kapitel vorgestellten Klassen `boost::intrusive::list` und `boost::intrusive::set` stellt Boost.Intrusive weitere zum Teil sehr spezielle Container zur Verfügung. Die wichtigsten dieser Container sind neben oben vorgestellten `boost::intrusive::slist` für einfach verkettete Listen und `boost::intrusive::unordered_set` für Hash-Container.

Kapitel 19

Boost.MultiArray

Boost.MultiArray ist eine Bibliothek, die den Umgang mit mehrdimensionalen Arrays vereinfacht. Sie bietet vor allem den Vorteil, dass sich ein mehrdimensionales Array wie ein Container aus der Standardbibliothek verwenden lässt. So stehen zum Beispiel Methoden wie `begin()` und `end()` zur Verfügung, um über Iteratoren auf Elemente in mehrdimensionalen Arrays zuzugreifen. So muss nicht mehr wie bei herkömmlichen C-Arrays mit Zeigern hantiert werden, was vor allem bei Arrays mit vielen Dimensionen unübersichtlich werden kann.

Beispiel 19.1 Eindimensionales Array mit `boost::multi_array`

```
#include <boost/multi_array.hpp>
#include <iostream>

int main()
{
  boost::multi_array<char, 1> a{boost::extents[6]};

  a[0] = 'B';
  a[1] = 'o';
  a[2] = 'o';
  a[3] = 's';
  a[4] = 't';
  a[5] = '\0';

  std::cout << a.origin() << '\n';
}
```

Um Arrays mit Boost.MultiArray zu erstellen, muss auf die Klasse `boost::multi_array` zugegriffen werden. Diese ist die wichtigste der von dieser Bibliothek angebotenen Klassen. Sie ist in der Headerdatei `boost/multi_array.hpp` definiert.
`boost::multi_array` ist ein Template, das zwei Parameter erwartet: Als ersten Parameter geben Sie den Typ an, den Sie im Array speichern wollen. Der zweite Parameter legt fest, wie viele Dimensionen das Array erhalten soll.
Beachten Sie, dass der zweite Parameter ausschließlich die Anzahl der Dimensionen festlegt – nicht die Anzahl der Elemente pro Dimension. Für Beispiel 19.1 bedeutet dies, dass es sich beim Array **a** um ein eindimensionales Array handelt.
Die Anzahl der Elemente pro Dimension wird zur Laufzeit festgelegt. So wird im Beispiel 19.1 auf das von Boost.MultiArray zur Verfügung gestellte globale Objekt **`boost::extents`** zugegriffen, mit dem Dimensionen eine Größe vorgegeben wird. Dieses Objekt wird dazu an den Konstruktor von **a** übergeben.
Ein Objekt vom Typ `boost::multi_array` kann grundsätzlich wie ein herkömmliches C-Array verwendet werden. So kann über `operator[]` und einen Index auf Elemente zugegriffen werden. Im Beispiel 19.1 werden, da es sich bei **a** um ein eindimensionales Array mit sechs Elementen handelt, fünf Buchstaben gefolgt vom Null-Zeichen gespeichert. Über die Methode `origin()` wird ein Zeiger auf das erste Elemente erhalten, der es ermöglicht, das im Array gespeicherte Wort `Boost` auszugeben.
Beachten Sie, dass beim Zugriff auf `operator[]` im Gegensatz zu anderen bekannten Containern aus der Standardbibliothek die Gültigkeit von Indizes überprüft wird. Bei einem ungültigen Index wird Ihr Programm mit `std::abort()` beendet. Möchten Sie, dass `boost::multi_array` keine Überprüfung auf Gültigkeit durchführt, müssen Sie das Macro `BOOST_DISABLE_ASSERTS` definieren, bevor Sie `boost/multi_array.hpp`

einbinden.

Beispiel 19.2 Ansichten und Subarrays eines zweidimensionalen Arrays

```
#include <boost/multi_array.hpp>
#include <algorithm>
#include <iostream>
#include <cstring>

int main()
{
  boost::multi_array<char, 2> a{boost::extents[2][6]};

  typedef boost::multi_array<char, 2>::array_view<1>::type array_view;
  typedef boost::multi_array_types::index_range range;
  array_view view = a[boost::indices[0][range{0, 5}]];

  std::memcpy(view.origin(), "tsooB", 6);
  std::reverse(view.begin(), view.end());

  std::cout << view.origin() << '\n';

  boost::multi_array<char, 2>::reference subarray = a[1];
  std::memcpy(subarray.origin(), "C++", 4);

  std::cout << subarray.origin() << '\n';
}
```

Im Beispiel 19.2 wird ein zweidimensionales Array erstellt. Die Anzahl der Elemente wird für die erste Dimension auf 2 und für die zweite Dimension auf 6 gesetzt. Sie können sich das Array als eine Tabelle vorstellen, die aus zwei Zeilen und sechs Spalten besteht.
In der ersten Zeile der Tabelle soll das Wort Boost gespeichert werden. Da für dieses Wort nur fünf Zeichen gespeichert werden müssen, wird eine *Ansicht* auf das Array erstellt, die genau fünf Stellen aus dem gesamten Array umfasst.
Eine Ansicht basiert auf der Klasse `boost::multi_array::array_view`. Es handelt sich hierbei um einen Typ, der von `boost::multi_array` abhängt, da eine Ansicht einen Teilausschnitt eines bestehenden Arrays beschreibt. Sie können über eine Ansicht auf einen bestimmten Teil eines Arrays zugreifen und diesen Teil so behandeln, als würde es sich um ein eigenständiges Array handeln.
`boost::multi_array::array_view` ist ein Template und erwartet als Template-Parameter die Anzahl der Dimensionen der Ansicht. Da im Beispiel 19.2 1 übergeben wird, das Array **a** jedoch aus zwei Dimensionen besteht, wird die neu zu erstellende Ansicht eine Dimension nicht berücksichtigen. Um das Wort Boost zu speichern, wird nur ein eindimensionales Array benötigt – weitere Dimensionen würden nur stören.
Während die Anzahl der Dimensionen als Template-Parameter angegeben wird, muss die Ausdehnung jeder Dimension zur Laufzeit festgelegt werden. Dies geschieht bei `boost::multi_array::array_view` nicht über **`boost::extents`**, sondern über **`boost::indices`**. Auch hierbei handelt es sich um ein globales Objekt, das von Boost.MultiArray zur Verfügung gestellt wird.
Ähnlich wie bei **`boost::extents`** müssen hinter **`boost::indices`** Indizes angegeben werden. Während bei **`boost::extents`** ausschließlich Zahlen angegeben werden dürfen, akzeptiert **`boost::indices`** auch Bereiche. Diese werden über die Klasse `boost::multi_array_types::index_range` definiert.
Der erste an **`boost::indices`** übergebene Parameter ist kein Bereich, sondern die Zahl 0. Wird eine Zahl angegeben, darf nicht auf `boost::multi_array_types::index_range` zugegriffen werden. **`boost::indices`** interpretiert diese Angabe derart, dass aus allen Elementen der ersten Dimension im Array **a** die Ansicht nur das erste Element zur Verfügung stellen soll – das mit dem Index 0.
Für den zweiten Parameter wird auf `boost::multi_array_types::index_range` zugegriffen, um einen Bereich zu definieren. Indem 0 und 5 an den Konstruktor dieser Klasse übergeben werden, wird auf die ersten fünf Elemente aus der ersten Dimension im Array **a** zugegriffen. Die Angaben werden derart interpretiert, dass der Bereich bei Index 0 beginnt und bei Index 5 endet, wobei Index 5 nicht mehr zum Bereich gehört. Das sechste Element aus der ersten Dimension wird demnach ignoriert.
Die Ansicht **`view`** stellt ein eindimensionales Array zur Verfügung, das aus fünf Elementen besteht. Es handelt sich hierbei um die ersten fünf Elemente in der ersten Zeile des Arrays **a**. Wenn im Folgenden auf **`view`** zugegriffen wird, um mit `std::memcpy()` eine Zeichenkette zu kopieren und die Elemente in der Ansicht mit `std::reverse()` umzudrehen, spielt dieser Zusammenhang keine Rolle. Ist eine Ansicht einmal erstellt, kann sehr

einfach auf einen Teil eines größeren Arrays zugegriffen werden. Die Ansicht verhält sich wie ein eigenes Array. Wenn mit `operator[]` direkt auf ein Array vom Typ `boost::multi_array` zugegriffen wird, hängt der Rückgabewert von der Anzahl der Dimensionen ab. So konnte im Beispiel 19.1 direkt auf einzelne char-Elemente im Array zugegriffen werden, weil das Array eindimensional war.
Im Beispiel 19.2 handelt es sich bei **a** um ein zweidimensionales Array. Bei einem Zugriff über den Operator `operator[]` wird nun nicht mehr ein char-Element zurückgegeben, sondern ein Subarray. Der entsprechende Typ des Subarrays wird von Boost.MultiArray nicht öffentlich gemacht, so dass auf `boost::multi_array::reference` zugegriffen werden muss. Bei diesem Typ handelt es sich jedoch nicht um `boost::multi_array::array_view`, auch wenn sich ein Subarray wie eine Ansicht verhält. Eine Ansicht muss explizit definiert werden und kann beliebige Teilbereiche eines Arrays umfassen, während ein Subarray automatisch bei einem Zugriff mit `operator[]` zurückgegeben wird und jeweils alle Elemente in jeder Dimension umfasst.

Beispiel 19.3 C-Array mit `boost::multi_array_ref` als MultiArray behandeln

```
#include <boost/multi_array.hpp>
#include <algorithm>
#include <iostream>
#include <cstring>

int main()
{
  char c[12] =
  {
    't', 's', 'o', 'o', 'B', '\0',
    'C', '+', '+', '\0', '\0', '\0'
  };

  boost::multi_array_ref<char, 2> a{c, boost::extents[2][6]};

  typedef boost::multi_array<char, 2>::array_view<1>::type array_view;
  typedef boost::multi_array_types::index_range range;
  array_view view = a[boost::indices[0][range{0, 5}]];

  std::reverse(view.begin(), view.end());
  std::cout << view.origin() << '\n';

  boost::multi_array<char, 2>::reference subarray = a[1];
  std::cout << subarray.origin() << '\n';
}
```

Boost.MultiArray stellt mit `boost::multi_array_ref` eine Klasse zur Verfügung, um ein MultiArray über ein bereits bestehendes C-Array zu stülpen. So stellt im Beispiel 19.3 **a** die von `boost::multi_array` bekannte Schnittstelle zur Verfügung, ohne Speicherplatz zu reservieren. Mit `boost::multi_array_ref` kann demnach ein C-Array – egal, aus wie vielen Dimensionen es besteht – wie ein mehrdimensionales Array vom Typ `boost::multi_array` behandelt werden. Dazu muss das C-Array lediglich als zusätzlicher Parameter neben dem bereits bekannten Objekt **`boost::extents`** an den Konstruktor übergeben werden.
Neben `boost::multi_array_ref` stellt Boost.MultiArray mit `boost::const_multi_array_ref` auch eine Klasse zur Verfügung, um ein C-Array als ein konstantes mehrdimensionales Array zu behandeln.

Kapitel 20

Boost.Container

Boost.Container ist eine Boost-Bibliothek, die die gleichen Container wie die Standardbibliothek anbietet. Dabei legt Boost.Container Wert auf zusätzliche Flexibilität. So können zum Beispiel alle Container dieser Bibliothek mit Boost.Interprocess in Shared Memory verwendet werden – etwas, was mit Containern aus der Standardbibliothek nicht zwangsläufig möglich ist.
Boost.Container bietet weitere Vorteile:

- Die Schnittstellen der Container stehen denen aus der C++11-Standardbibliothek in nichts nach. So können Sie auch in C++98-Programmen zum Beispiel auf Methoden wie `emplace_back()` zugreifen, die erst mit C++11 zur Standardbibliothek hinzugefügt wurden.
- Mit `boost::container::slist` oder `boost::container::stable_vector` werden zusätzliche Container angeboten, die in der Standardbibliothek fehlen.
- Die Implementation ist plattformunabhängig. Die Container verhalten sich überall gleich, so dass Sie sich nicht um mögliche Unterschiede zwischen Implementationen der Standardbibliothek kümmern müssen.
- Die Container von Boost.Container unterstützen *unvollständige Typen* und können verwendet werden, um rekursive Container zu definieren.

Beispiel 20.1 zeigt, was mit unvollständigen Typen gemeint ist.

Beispiel 20.1 Rekursive Container mit Boost.Container

```
#include <boost/container/vector.hpp>

using namespace boost::container;

struct animal
{
  vector<animal> children;
};

int main()
{
  animal parent, child1, child2;
  parent.children.push_back(child1);
  parent.children.push_back(child2);
}
```

Die Klasse `animal` besitzt eine Eigenschaft **`children`**, deren Typ boost::container::vector<animal> ist. `boost::container::vector` ist in der Headerdatei `boost/container/vector.hpp` definiert. Der Typ der Eigenschaft **`children`** basiert demnach auf der Klasse `animal`, in der **`children`** definiert ist. Zum Zeitpunkt des Zugriffs ist `animal` nicht vollständig definiert. Während der Standard für Container aus der Standardbibliothek keine explizite Unterstützung für unvollständige Typen vorsieht, sind rekursive Container mit Boost.Container explizit möglich. Es hängt von der Implementation der Standardbibliothek ab, ob dies auch mit Containern aus dieser möglich ist.

Beispiel 20.2 `boost::container::stable_vector` in Aktion

```
#include <boost/container/stable_vector.hpp>
#include <iostream>

using namespace boost::container;

int main()
{
  stable_vector<int> v(2, 1);
  int &i = v[1];
  v.erase(v.begin());
  std::cout << i << '\n';
}
```

Boost.Container bietet neben den aus der Standardbibliothek bekannten Containern weitere an. So wird im Beispiel 20.2 mit `boost::container::stable_vector` ein Container vorgestellt, der sich ähnlich wie `std::vector` verhält. Werden `boost::container::stable_vector` Elemente hinzugefügt oder werden Elemente entfernt, bleiben jedoch Iteratoren und Referenzen auf bestehende Elemente gültig. Dies ist möglich, weil die Elemente im `boost::container::stable_vector` nicht in einem zusammenhängenden Speicherblock liegen. Es kann weiterhin mit einem Index auf Elemente zugegriffen werden. Elemente liegen jedoch nicht direkt nebeneinander im Speicher.
Boost.Container garantiert, dass die Referenz **`i`** im Beispiel 20.2 gültig bleibt, wenn das erste Element im Vektor gelöscht wird. Das Beispielprogramm gibt demnach `1` aus.
`boost::container::stable_vector` ist in der Headerdatei `boost/container/stable_vector.hpp` definiert.
Weitere von Boost.Container angebotene Container sind `boost::container::flat_set`, `boost::container::flat_map`, `boost::container::slist` und `boost::container::static_vector`:

- `boost::container::flat_set` und `boost::container::flat_map` ähneln `std::set` und `std::map`, sind jedoch nicht als Baum, sondern als sortierte Vektoren implementiert. So ist ein schnellerer Zugriff auf wie auch eine schnellere Iteration über Elemente möglich. Das Einfügen und Entfernen von Elementen ist jedoch langsamer.

  Die beiden Container sind in den Headerdateien `boost/container/flat_set.hpp` und `boost/container/flat_map.hpp` definiert.

- `boost::container::slist` ist eine einfach verkettete Liste. Sie ähnelt dem Container `std::forward_list`, der in C++11 neu in die Standardbibliothek aufgenommen wurde. `boost::container::slist` bietet die Methode `size()` an, die bei `std::forward_list` fehlt.

  `boost::container::slist` ist in der Headerdatei `boost/container/slist.hpp` definiert.

- `boost::container::static_vector` speichert Elemente ähnlich wie `std::array` direkt im Container. Ähnlich wie `std::array` besitzt der Container eine konstante Kapazität. Die Kapazität sagt jedoch nichts über die Anzahl der Elemente im Container aus. Mit `push_back()`, `pop_back()`, `insert()` und `erase()` stehen mehrere Methoden zur Verfügung, um Elemente hinzuzufügen oder zu entfernen. In dieser Hinsicht ähnelt `boost::container::static_vector` `std::vector`. So existiert mit `size()` auch eine Methode, um die Anzahl der momentan im Container gespeicherten Elemente zu erhalten.

  Die Kapazität ist konstant, kann jedoch mit `resize()` neu gesetzt werden. `push_back()` ändert die Kapazität jedoch nicht. Sie dürfen mit `push_back()` nur dann ein Element hinzufügen, wenn die Kapazität größer als die Anzahl der momentan gespeicherten Elemente ist. Andernfalls wirft `push_back()` eine Ausnahme vom Typ `std::bad_alloc`.

  `boost::container::static_vector` ist in der Headerdatei `boost/container/static_vector.hpp` definiert.

Teil IV

Datenstrukturen

Datenstrukturen sind container-ähnliche Objekte, die ein oder mehrere Elemente speichern können. Sie unterscheiden sich von Containern, da sie bestimmten Einschränkungen unterliegen. So ist es zum Beispiel mit einigen Datenstrukturen, die im Folgenden vorgestellt werden, nicht möglich, in einer einzigen Iteration auf alle Elemente zuzugreifen.

- Boost.Optional macht es einfach, optionale Rückgabewerte zu kennzeichnen. Objekte, die mit Boost.Optional erstellt werden, sind entweder leer oder enthalten genau ein Element. So muss nicht auf spezielle Werte wie -1 oder einen Nullzeiger zugegriffen werden, um anzugeben, dass eine Funktion keinen Wert zurückgibt.
- Boost.Tuple bietet mit `boost::tuple` eine Klasse an, die seit C++11 auch von der Standardbibliothek angeboten wird.
- Mit Boost.Any und Boost.Variant können Variablen erstellt werden, die Werte unterschiedlicher Typen speichern können. Während Boost.Any tatsächlich jeden beliebigen Typ unterstützt, werden die von einer Boost.Variant-Variablen zu unterstützenden Typen als Template-Parameter angegeben.
- Boost.PropertyTree bietet eine Baumstruktur an. Die Bibliothek wird üblicherweise verwendet, um Konfigurationsdaten in einem Programm zu verwalten. Es ist möglich, die Baumstruktur zu serialisieren und in einer Datei zu speichern.
- Boost.DynamicBitset bietet eine Klasse an, die `std::bitset` ähnelt, jedoch zur Laufzeit konfiguriert wird.
- Die von Boost.Tribool angebotene Datenstruktur ähnelt bool, kennt jedoch drei Status.
- Boost.CompressedPair definiert mit `boost::compressed_pair` eine Klasse, die `std::pair` ersetzen kann und die sogenannte empty base class optimization unterstützt.

Kapitel 21

Boost.Optional

Die Bibliothek Boost.Optional bietet mit `boost::optional` eine Klasse an, die für optionale Rückgabewerte verwendet werden kann. Damit sind Rückgabewerte von Funktionen gemeint, die unter Umständen kein Ergebnis zurückgeben können. Sehen Sie sich dazu Beispiel 21.1 an, in dem eine herkömmliche Lösung ohne Boost.Optional zum Tragen kommt.

Beispiel 21.1 Spezielle Werte zur Kennzeichnung optionaler Rückgabewerte

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

int get_even_random_number()
{
  int i = std::rand();
  return (i % 2 == 0) ? i : -1;
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  int i = get_even_random_number();
  if (i != -1)
    std::cout << std::sqrt(static_cast<float>(i)) << '\n';
}
```

Im Beispiel 21.1 wird eine Funktion `get_even_random_number()` verwendet, die eine gerade Zufallszahl zurückgeben soll. Dazu wird auf die Funktion `std::rand()` aus der Standardbibliothek zugegriffen. Die Implementation von `get_even_random_number()` ist recht naiv: Gibt `std::rand()` eine gerade Zufallszahl zurück, wird diese als Rückgabewert von `get_even_random_number()` weitergereicht. Ist die von `std::rand()` generierte Zufallszahl ungerade, wird -1 zurückgegeben.
So wie `get_even_random_number()` implementiert ist, bedeutet -1, dass keine gerade Zufallszahl generiert werden konnte und zurückgegeben werden kann. Die Funktion `get_even_random_number()` kann nicht garantieren, dass eine gerade Zufallszahl zurückgegeben wird. Der Rückgabewert ist optional.
Viele Funktionen verwenden spezielle Werte wie -1, um anzugeben, dass kein Ergebnis zurückgegeben werden kann. So gibt zum Beispiel die Methode `find()` der Klasse `std::string` den speziellen Wert **`std::string::npos`** zurück, wenn ein Substring nicht gefunden werden kann. Funktionen, deren Rückgabewert ein Zeiger ist, geben oft 0 zurück, um dem Aufrufer mitzuteilen, dass kein Ergebnis existiert.
Boost.Optional bietet mit `boost::optional` eine Klasse an, die optionale Rückgabewerte deutlich macht.

Beispiel 21.2 Optionale Rückgabewerte mit `boost::optional`

```
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using boost::optional;
```

```
optional<int> get_even_random_number()
{
  int i = std::rand();
  return (i % 2 == 0) ? i : optional<int>{};
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  optional<int> i = get_even_random_number();
  if (i)
    std::cout << std::sqrt(static_cast<float>(*i)) << '\n';
}
```

Im Beispiel 21.2 hat die Funktion `get_even_random_number()` mit boost::optional<int> einen neuen Typ für den Rückgabewert erhalten. Bei `boost::optional` handelt es sich um ein Template, das mit dem eigentlichen Typ des Rückgabewerts instanziiert werden muss. Um `boost::optional` verwenden zu können, muss außerdem die Headerdatei `boost/optional.hpp` eingebunden werden.
Generiert `get_even_random_number()` eine gerade Zufallszahl, wird diese mit `return` direkt zurückgeben. Sie wird automatisch in ein Objekt vom Typ boost::optional<int> gepackt, da `boost::optional` einen entsprechenden nicht-exklusiven Konstruktor anbietet. Wird keine gerade Zufallszahl generiert, wird ein leeres Objekt vom Typ boost::optional<int> zurückgegeben. Dieses wird über den Aufruf des Standardkonstruktors erstellt.
In der Funktion `main()` wird überprüft, ob **`i`** nicht leer ist. Ist dies der Fall, wird über `operator*` auf die Zahl zugegriffen, die in **`i`** gespeichert ist. `boost::optional` scheint daher ähnlich wie ein Zeiger zu funktionieren. Es ist jedoch wichtig, `boost::optional` nicht mit einem Zeiger gleichzusetzen, da zum Beispiel der Copy-Konstruktor bei `boost::optional` den im Objekt gespeicherten Wert kopiert, während ein Zeiger nicht den Wert kopiert, auf den er zeigt.

Beispiel 21.3 Verschiedene Methoden von `boost::optional`

```
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using boost::optional;

optional<int> get_even_random_number()
{
  int i = std::rand();
  return optional<int>{i % 2 == 0, i};
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  optional<int> i = get_even_random_number();
  if (i.is_initialized())
    std::cout << std::sqrt(static_cast<float>(i.get())) << '\n';
}
```

Beispiel 21.3 stellt verschiedene Methoden von `boost::optional` vor. So bietet die Klasse einen speziellen Konstruktor an, dem als ersten Parameter eine Bedingung übergeben werden kann. Ist die Bedingung wahr, wird ein Objekt vom Typ `boost::optional` mit dem zweiten Parameter initialisiert. Ist die Bedingung falsch, wird ein leeres Objekt vom Typ `boost::optional` erstellt. Dieser Konstruktor wird im Beispiel 21.3 in der Funktion `get_even_random_number()` verwendet.
Über `is_initialized()` kann explizit überprüft werden, ob ein Objekt vom Typ `boost::optional` nicht leer ist. Boost.Optional spricht von initialisierten und nicht-initialisierten Objekten – daher der Name der Methode `is_initialized()`. Die Methode `get()` wiederum ist gleichbedeutend mit `operator*`.

Beispiel 21.4 Verschiedene Hilfsfunktionen von Boost.Optional

```
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace boost;

optional<int> get_even_random_number()
{
  int i = std::rand();
  return make_optional(i % 2 == 0, i);
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  optional<int> i = get_even_random_number();
  double d = get_optional_value_or(i, 0);
  std::cout << std::sqrt(d) << '\n';
}
```

Boost.Optional bietet freistehende Hilfsfunktionen wie `boost::make_optional()` und `boost::get_optional_value_or()` an. Während `boost::make_optional()` verwendet werden kann, um ein Objekt vom Typ `boost::optional` zu erzeugen, kann `boost::get_optional_value_or()` aufgerufen werden, wenn ein alternativer Wert verwendet werden soll, sollte `boost::optional` leer sein.
Die Funktion `get_optional_value_or()` wird auch als Methode von `boost::optional` angeboten. Sie heißt dann `get_value_or()`.
Boost.Optional bietet mit `boost/optional/optional_io.hpp` auch eine Headerdatei an, die Stream-Operatoren überlädt, um Objekte vom Typ `boost::optional` zum Beispiel direkt auf die Standardausgabe ausgeben zu können.

Kapitel 22

Boost.Tuple

Die Bibliothek Boost.Tuple stellt eine Klasse `boost::tuple` zur Verfügung, die als eine verallgemeinerte Version von `std::pair` bezeichnet werden kann. Während `std::pair` genau zwei Werte speichert, kann `boost::tuple` beliebig viele Werte speichern.
Seit C++11 bietet die Standardbibliothek die Klasse `std::tuple` an. Arbeiten Sie in einer C++11-Entwicklungsumgebung, können Sie Boost.Tuple ignorieren, da `boost::tuple` und `std::tuple` gleich sind.

Beispiel 22.1 `boost::tuple` als `std::pair`-Ersatz

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int> animal;
  animal a{"cat", 4};
  std::cout << a << '\n';
}
```

Um die Klasse `boost::tuple` verwenden zu können, muss die Headerdatei `boost/tuple/tuple.hpp` eingebunden werden. Möchten Sie Variablen vom Typ `boost::tuple` von Streams lesen und auf Streams ausgeben, müssen Sie außerdem die Headerdatei `boost/tuple/tuple_io.hpp` einbinden. Boost.Tuple stellt keine Master-Headerdatei zur Verfügung, die automatisch alle anderen Headerdateien einbindet.
`boost::tuple` wird grundsätzlich genauso verwendet wie `std::pair`. So können Sie wie im Beispiel 22.1 zwei Template-Parameter angeben, um zwei Werte zu speichern – in diesem Fall einen vom Typ `std::string` und einen vom Typ int. Die Typdefinition heißt animal, weil damit der Name und die Anzahl der Beine von Tieren angegeben werden können sollen.
Während die obige Typdefinition von animal auch mit `std::pair` hätte erfolgen können, ist es möglich, Objekte vom Typ `boost::tuple` in einen Stream zu schreiben. Dazu muss wie bereits erwähnt die Headerdatei `boost/tuple/tuple_io.hpp` eingebunden werden, in der sich die Definitionen der entsprechenden Stream-Operatoren befinden. So gibt Beispiel 22.1 `(cat 4)` aus.

Beispiel 22.2 `boost::tuple` als das bessere `std::pair`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a{"cat", 4, true};
  std::cout << std::boolalpha << a << '\n';
}
```

Beispiel 22.2 unterstützt zusätzlich zum Namen und zur Anzahl der Beine einen Wahrheitswert, der angibt, ob ein Tier einen Schwanz hat. Alle drei Werte können in einem Tuple untergebracht werden. Wenn Sie das Programm ausführen, wird `(cat 4 true)` ausgegeben.
So wie es zu `std::pair` eine Hilfsfunktion `std::make_pair()` gibt, kann ein Tuple mit Hilfe einer Funktion `boost::make_tuple()` erstellt werden. Sehen Sie sich dazu Beispiel 22.3 an.

Beispiel 22.3 Tuple erstellen mit `boost::make_tuple()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <iostream>

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << boost::make_tuple("cat", 4, true) << '\n';
}
```

Es ist auch möglich, ein Tuple zu erstellen, das Referenzen enthält. Sehen Sie sich dazu Beispiel 22.4 an.

Beispiel 22.4 Tuple mit Referenzen

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <boost/ref.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "cat";
  std::cout.setf(std::ios::boolalpha);
  std::cout << boost::make_tuple(boost::ref(s), 4, true) << '\n';
}
```

Während die Werte 4 und `true` als Kopie übergeben und direkt im Tuple gespeichert werden, ist das erste Element eine Referenz auf den String **`s`**. Um eine derartige Referenz zu erstellen, wird auf `boost::ref()` aus Boost.Ref zugegriffen. Entsprechend muss auf `boost::cref()` zugegriffen werden, wenn eine konstante Referenz erstellt werden soll.
Sie dürfen anstatt `boost::ref()` `std::ref()` aus der C++11-Standardbibliothek verwenden. Im obigen Beispiel muss jedoch `boost::ref()` zum Einsatz kommen, weil nur Boost.Ref einen Stream-Operator zur direkten Ausgabe auf die Standardausgabe unterstützt.
Nachdem Sie gesehen haben, wie Tuple erstellt werden, wird im Folgenden auf Elemente eines Tuples zugegriffen. Bei `std::pair` geschieht dies über die Eigenschaften **`first`** und **`second`**. Da ein Tuple jedoch keine festgelegte Anzahl an Elementen besitzt, findet der Zugriff auf eine andere Art und Weise statt.

Beispiel 22.5 Lesender Zugriff auf Werte in einem Tuple

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a = boost::make_tuple("cat", 4, true);
  std::cout << a.get<0>() << '\n';
  std::cout << boost::get<0>(a) << '\n';
}
```

Es gibt zwei Möglichkeiten, auf ein Element in einem Tuple zuzugreifen: Entweder wird für das Tuple die Methode `get()` aufgerufen oder das Tuple als einziger Parameter an eine freistehende Funktion `boost::get()` übergeben. In beiden Fällen muss der Index des entsprechenden Elements im Tuple als Template-Parameter übergeben werden. Für Beispiel 22.5 bedeutet das, dass in beiden Fällen auf das erste Element im Tuple **`a`** zugegriffen wird und zweimal `cat` ausgegeben wird.

Wird ein ungültiger Index angegeben, kompiliert der Code nicht. Die Gültigkeit der Indizes wird zur Kompilierung überprüft, so dass es nicht zu Laufzeitfehlern kommen kann.
Um einen Wert in einem Tuple zu ändern, müssen Sie ebenfalls entweder auf die Methode `get()` oder die freistehende Funktion `boost::get()` zugreifen.

Beispiel 22.6 Schreibender Zugriff auf Werte in einem Tuple

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a = boost::make_tuple("cat", 4, true);
  a.get<0>() = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

Sowohl `get()` als auch `boost::get()` geben eine Referenz zurück. Im Beispiel 22.6 wird entsprechend der Name geändert und vom Programm `(dog 4 true)` ausgegeben.
Boost.Tuple stellt nicht nur überladene Operatoren zur Verfügung, um Tuple auf einen Stream auszugeben oder von einem Stream zu lesen. Boost.Tuple definiert auch Vergleichsoperatoren. Wollen Sie Tuple vergleichen, müssen Sie jedoch mit `boost/tuple/tuple_comparison.hpp` eine zusätzliche Headerdatei einbinden.

Beispiel 22.7 Tuple vergleichen

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a1 = boost::make_tuple("cat", 4, true);
  animal a2 = boost::make_tuple("shark", 0, true);
  std::cout << std::boolalpha << (a1 != a2) << '\n';
}
```

Beispiel 22.7 gibt `true` aus, da die beiden Tuple **`a1`** und **`a2`** verschieden sind.
Die Headerdatei `boost/tuple/tuple_comparison.hpp` enthält auch Definitionen anderer Vergleichsoperatoren wie `operator>`, die einen lexikographischen Vergleich durchführen.
Boost.Tuple unterstützt eine besondere Form von Tuple namens *Tier*. Ein Tier ist ein Tuple, dessen Elemente alle Referenzen sind. Es kann mit der Funktion `boost::tie()` erstellt werden.

Beispiel 22.8 Ein Tier mit `boost::tie()` erstellen

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, int&, bool&> animal;
  std::string name = "cat";
  int legs = 4;
  bool tail = true;
  animal a = boost::tie(name, legs, tail);
  name = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

Im Beispiel 22.8 wird ein Tier **`a`** erstellt, das aus Referenzen auf die Variablen **`name`**, **`legs`** und **`tail`** besteht. Indem die Variable **`name`** auf einen neuen Wert gesetzt wird, wird gleichzeitig das Tier geändert.

Anstatt `boost::tie()` zu verwenden, hätte Beispiel 22.8 auch mit `boost::make_tuple()` und `boost::ref()` geschrieben werden können. Sehen Sie sich dazu Beispiel 22.9 an.

Beispiel 22.9 Ein Tier ohne `boost::tie()` erstellen

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, int&, bool&> animal;
  std::string name = "cat";
  int legs = 4;
  bool tail = true;
  animal a = boost::make_tuple(boost::ref(name), boost::ref(legs),
    boost::ref(tail));
  name = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

`boost::tie()` verkürzt die Schreibweise. Die Funktion bietet sich auch an, wenn Werte aus einem Tuple entpackt werden soll. Sehen Sie sich dazu Beispiel 22.10 an, in dem eine Funktion ein Tuple zurückgibt und die einzelnen Werte im Tuple sofort in Variablen gespeichert werden.

Beispiel 22.10 Rückgabewerte einer Funktion aus einem Tuple entpacken

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

boost::tuple<std::string, int> new_cat()
{
  return boost::make_tuple("cat", 4);
}

int main()
{
  std::string name;
  int legs;
  boost::tie(name, legs) = new_cat();
  std::cout << name << ", " << legs << '\n';
}
```

Mit Hilfe von `boost::tie()` werden der String „cat“ und die Zahl 4, die beide zusammen in einem Tuple von `new_cat()` zurückgegeben werden, direkt in den Variablen **`name`** und **`legs`** gespeichert.

Kapitel 23

Boost.Any

In strengtypisierten Sprachen wie C++ hat jede Variable einen ganz bestimmten Typ. Dieser Typ entscheidet darüber, welche Art von Daten in einer Variablen gespeichert werden können. Wer in einer Variablen eine beliebige Art von Daten speichern möchte, muss eine Programmiersprache wie beispielsweise JavaScript verwenden: Dort kann in einer Variablen ein String, danach eine Zahl und dann ein Wahrheitswert gespeichert werden. In JavaScript kann jede Variable Daten beliebiger Art speichern.
Die Bibliothek Boost.Any stellt eine Klasse `boost::any` zur Verfügung, die es ähnlich wie in JavaScript in C++ ermöglicht, Daten beliebiger Art in einer entsprechenden Variablen zu speichern.

Beispiel 23.1 `boost::any` in Aktion

```
#include <boost/any.hpp>

int main()
{
  boost::any a = 1;
  a = 3.14;
  a = true;
}
```

Um `boost::any` verwenden zu können, muss die Headerdatei `boost/any.hpp` eingebunden werden. Sie können dann Objekte vom Typ `boost::any` erstellen, in denen Sie Daten beliebiger Art speichern können. So wird im Beispiel 23.1 in **a** zuerst ein int-Wert, dann eine Kommazahl vom Typ double und dann ein Wahrheitswert gespeichert.
Beachten Sie, dass es Mindestanforderungen gibt. So muss der Typ des Werts, der gespeichert werden soll, einen Copy-Konstruktor besitzen. Dies bedeutet beispielsweise, dass es nicht möglich ist, C-Arrays in einer `boost::any`-Variablen zu speichern. Denn diese besitzen keinen Copy-Konstruktor.
Möchten Sie einen String in einer `boost::any`-Variablen speichern und nicht lediglich einen Zeiger auf einen C-String, greifen Sie wie im Beispiel 23.2 explizit auf `std::string` zu.

Beispiel 23.2 Einen String in `boost::any` speichern

```
#include <boost/any.hpp>
#include <string>

int main()
{
  boost::any a = std::string{"Boost"};
}
```

Um auf den Wert einer `boost::any`-Variablen zuzugreifen, muss der Cast-Operator `boost::any_cast` verwendet werden. Sehen Sie sich dazu Beispiel 23.3 an.

Beispiel 23.3 Zugriff auf Werte mit `boost::any_cast`

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  boost::any a = 1;
```

```
  std::cout << boost::any_cast<int>(a) << '\n';
  a = 3.14;
  std::cout << boost::any_cast<double>(a) << '\n';
  a = true;
  std::cout << std::boolalpha << boost::any_cast<bool>(a) << '\n';
}
```

Indem der entsprechende Typ als Template-Parameter an `boost::any_cast` übergeben wird, wird der Wert der `boost::any`-Variablen umgewandelt. Sollte ein ungültiger Typ angegeben werden und eine entsprechende Konvertierung nicht durchgeführt werden können, wird eine Ausnahme vom Typ `boost::bad_any_cast` geworfen.

Beispiel 23.4 `boost::bad_any_cast` bei fehlerhaftem Zugriff

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::any a = 1;
    std::cout << boost::any_cast<float>(a) << '\n';
  }
  catch (boost::bad_any_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Im Beispiel 23.4 wird eine Ausnahme geworfen, weil der als Template-Parameter übergebene Typ float nicht dem Typ int entspricht, der in der Variablen **a** verwendet wird. Es ist wichtig, jeweils genau den Typ als Template-Parameter anzugeben, der in der entsprechenden `boost::any`-Variablen verwendet wird. So würde Beispiel 23.4 auch dann eine Ausnahme werfen, wenn als Template-Parameter short oder long angegeben werden würde.
Da die Klasse `boost::bad_any_cast` von `std::bad_cast` abgeleitet ist, können Sie in `catch`-Handlern auch Ausnahmen dieses Typs abfangen.
Möchten Sie wissen, ob in einer Variablen vom Typ `boost::any` ein Wert gespeichert ist, können Sie die Methode `empty()` aufrufen. Um zu erfahren, welchen Typ ein in `boost::any` gespeicherter Wert besitzt, können Sie `type()` verwenden.

Beispiel 23.5 Momentan gespeicherten Typ abfragen

```
#include <boost/any.hpp>
#include <typeinfo>
#include <iostream>

int main()
{
  boost::any a = 1;
  if (!a.empty())
  {
    const std::type_info &ti = a.type();
    std::cout << ti.name() << '\n';
  }
}
```

Im Beispiel 23.5 wird sowohl `empty()` als auch `type()` aufgerufen. Während `empty()` einen Wahrheitswert zurückgibt, ist der Typ des Rückgabewerts von `type()` die in der Headerdatei `typeinfo` definierte Klasse `std::type_info`.
Abschließend sehen Sie im Beispiel 23.6, wie Sie mit `boost::any_cast` einen Zeiger auf einen Wert in einer `boost::any`-Variable erhalten.

Beispiel 23.6 Zugriff auf Werte über einen Zeiger

```
#include <boost/any.hpp>
#include <iostream>
```

```
int main()
{
  boost::any a = 1;
  int *i = boost::any_cast<int>(&a);
  std::cout << *i << '\n';
}
```

Sie müssen lediglich einen Zeiger auf die `boost::any`-Variable als Parameter an `boost::any_cast` übergeben. Der Template-Parameter bleibt unverändert.

Kapitel 24

Boost.Variant

Boost.Variant bietet mit `boost::variant` einen Typ an, der `union` ähnelt. So können in einer `boost::variant`-Variablen Werte unterschiedlicher Typen gespeichert werden. Zu jedem Zeitpunkt kann lediglich ein Wert gespeichert sein. Wird einer `boost::variant`-Variablen ein neuer Wert zugewiesen, wird der alte überschrieben. Der neue Wert darf jedoch einen anderen Typ als der alte Wert haben. Einzige Voraussetzung ist, dass die verwendeten Typen als Template-Parameter an `boost::variant` übergeben wurden und der entsprechenden `boost::variant`-Variablen bekannt sind.
`boost::variant` unterstützt beliebige Typen. So kann zum Beispiel ein `std::string` in einer `boost::variant`-Variablen gespeichert werden – etwas, was mit `union` vor C++11 nicht möglich war. Mit C++11 wurden die Regeln für `union` aufgeweicht, so dass ein `std::string` Bestandteil einer `union` sein kann. Da die Initialisierung der `std::string`-Variablen in einer `union` ein placement new erfordert sowie der Destruktor explizit aufgerufen werden muss, kann der Einsatz von `boost::variant` auch in einer Entwicklungsumgebung, die C++11 unterstützt, sinnvoll sein.

Beispiel 24.1 `boost::variant` in Aktion

```
#include <boost/variant.hpp>
#include <string>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  v = 'A';
  v = "Boost";
}
```

Boost.Variant stellt eine Klasse `boost::variant` zur Verfügung, die in der Headerdatei `boost/variant.hpp` definiert ist. Da es sich bei `boost::variant` um ein Template handelt, muss mindestens ein Template-Parameter angegeben werden. Der oder die Template-Parameter beschreiben, welche Typen unterstützt werden. Für Beispiel 24.1 bedeutet dies, dass in der Variablen **v** ein double, ein char oder ein `std::string` gespeichert werden kann. Würden Sie versuchen, der Variablen **v** eine Zahl vom Typ int zuzuweisen, würde der Code nicht kompilieren.

Beispiel 24.2 Zugriff auf `boost::variant` mit `boost::get()`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << boost::get<double>(v) << '\n';
  v = 'A';
  std::cout << boost::get<char>(v) << '\n';
  v = "Boost";
  std::cout << boost::get<std::string>(v) << '\n';
}
```

Um die in der Variablen **v** gespeicherten Werte auszugeben, greifen Sie wie im Beispiel 24.2 auf eine freistehende Funktion `boost::get()` zu.
`boost::get()` erwartet als Template-Parameter einen der Typen, die für die entsprechende Variable erlaubt sind. Geben Sie einen ungültigen Typ an, führt dies zu einem Laufzeitfehler. Es findet keine Überprüfung der Typen zur Kompilierung statt.
Wenn Sie lediglich Werte auf einen Stream wie die Standardausgabe ausgeben möchten, können Sie die Gefahr falscher Typen insofern umgehen, als dass Sie Variablen vom Typ `boost::variant` direkt in einen Stream schreiben können. Sehen Sie sich dazu Beispiel 24.3 an.

Beispiel 24.3 Direkte Ausgabe von `boost::variant` auf einen Stream

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << v << '\n';
  v = 'A';
  std::cout << v << '\n';
  v = "Boost";
  std::cout << v << '\n';
}
```

Für einen typsicheren Zugriff auf in einer `boost::variant`-Variablen gespeicherte Werte können Sie die Funktion `boost::apply_visitor()` verwenden.

Beispiel 24.4 Einen Besucher für `boost::variant` erstellen

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

struct output : public boost::static_visitor<>
{
  void operator()(double d) const { std::cout << d << '\n'; }
  void operator()(char c) const { std::cout << c << '\n'; }
  void operator()(std::string s) const { std::cout << s << '\n'; }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output{}, v);
  v = 'A';
  boost::apply_visitor(output{}, v);
  v = "Boost";
  boost::apply_visitor(output{}, v);
}
```

`boost::apply_visitor()` wird als erster Parameter ein Objekt vom Typ einer Klasse übergeben, die von `boost::static_visitor` abgeleitet sein muss. Die Klasse muss für jeden Typ der `boost::variant`-Variablen, für die sie verwendet werden soll, den Operator `operator()` überladen. Im Beispiel 24.4 muss dieser Operator entsprechend dreimal überladen sein, weil **v** die Typen double, char und `std::string` unterstützt.
`boost::static_visitor` ist ein Template. Der Typ des Rückgabewerts der überladenen Operatoren muss als Template-Parameter angegeben werden. Besitzen wie im Beispiel 24.4 die Methoden keinen Rückgabewert, wird kein Template-Parameter angegeben.
Der zweite Parameter, der an `boost::apply_visitor()` übergeben wird, ist eine Variable vom Typ `boost::variant`.
Wird `boost::apply_visitor()` verwendet, wird automatisch der überladene Operator `operator()` aufgerufen, der zum Typ des momentan in der `boost::variant`-Variablen gespeicherten Werts passt. So wird im

Beispiel 24.4 bei jedem Aufruf von `boost::apply_visitor()` ein anderer überladener Operator aufgerufen – zuerst der für double, dann der für char und abschließend der für `std::string`.
Der Vorteil von `boost::apply_visitor()` ist, dass nicht nur automatisch der richtige Operator aufgerufen wird und der entsprechende Wert aus der `boost::variant`-Variablen als Parameter übergeben wird. `boost::apply_visitor()` stellt sicher, dass für alle von einer `boost::variant`-Variablen unterstützten Typen Operatoren überladen wurden. Würde im Beispiel 24.4 eine der drei Methoden fehlen, könnte der Code nicht kompiliert werden.
Wenn wie im vorherigen Beispiel mehrere überladene Operatoren das Gleiche tun, kann wie im Beispiel 24.5 der Code mit Hilfe eines Templates vereinfacht werden.

Beispiel 24.5 Ein Besucher mit Template-Funktion für `boost::variant`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

struct output : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const { std::cout << t << '\n'; }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output{}, v);
  v = 'A';
  boost::apply_visitor(output{}, v);
  v = "Boost";
  boost::apply_visitor(output{}, v);
}
```

Da bei `boost::apply_visitor()` zur Kompilierung sichergestellt wird, dass der Code korrekt ist, sollte diese Funktion `boost::get()` vorgezogen werden.

Kapitel 25

Boost.PropertyTree

Boost.PropertyTree bietet mit `boost::property_tree::ptree` eine Baumstruktur an, um Schüssel/Wert-Paare zu speichern. Baumstruktur bedeutet, dass es einen Stamm mit beliebig vielen Ästen gibt und jeder Ast wiederum beliebig viele Zweige haben kann. Sie kennen die Baumstruktur zum Beispiel von Dateisystemen, die ein Hauptverzeichnis mit Unterverzeichnissen haben, die ihrerseits wiederum Unterverzeichnisse haben – und so weiter.
Um die Klasse `boost::property_tree::ptree` verwenden zu können, müssen Sie die Headerdatei `boost/property_tree/ptree.hpp` einbinden. Da diese Headerdatei alle anderen von Boost.PropertyTree definierten Headerdateien einbindet, müssen Sie sich um keine weiteren Headerdateien kümmern.

Beispiel 25.1 Datenzugriff auf `boost::property_tree::ptree`

```
#include <boost/property_tree/ptree.hpp>
#include <iostream>

using boost::property_tree::ptree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");

  ptree &c = pt.get_child("C:");
  ptree &windows = c.get_child("Windows");
  ptree &system = windows.get_child("System");
  std::cout << system.get_value<std::string>() << '\n';
}
```

Im Beispiel 25.1 wird `boost::property_tree::ptree` verwendet, um einen Pfad auf ein Verzeichnis zu speichern. Dazu wird die Methode `put()` aufgerufen. Diese Methode erwartet zwei Parameter, da es sich bei `boost::property_tree::ptree` um eine Baumstruktur handelt, die Schüssel/Wert-Paare speichert. Der Baum besteht nicht nur aus Ästen und Zweigen – Ästen und Zweigen muss ein Wert zugewiesen werden. Im Beispiel 25.1 ist das die Angabe „20 files“.
Interessanter ist der erste an `put()` übergebene Parameter. Es handelt sich hierbei um einen Pfad auf ein Verzeichnis. Es wird jedoch nicht der unter Windows übliche Backslash verwendet, um Verzeichnisnamen zu trennen. Als Trennzeichen kommt ein Punkt zum Einsatz.
Es ist wichtig, dass der Punkt verwendet wird, da dieser das von Boost.PropertyTree definierte Trennzeichen für Schlüssel ist. Die Angabe „C:.Windows.System“ führt dazu, dass **`pt`** einen Ast namens C: erhält, der wiederum einen Ast namens Windows besitzt, der wiederum einen Ast namens System hat. Der Punkt stellt sicher, dass die gewünschte verästelte Struktur mit drei Ebenen entsteht. Wäre hingegen „C:\Windows\System“ als Schlüssel verwendet worden, hätte **`pt`** einen Ast namens C:\Windows\System erhalten, ohne dass dieser eigene Äste hätte.
Nach dem Aufruf von `put()` wird auf **`pt`** zugegriffen, um den gespeicherten Wert „20 files“ zu lesen und auf die Standardausgabe auszugeben. Dies geschieht, indem sich von Ast zu Ast – oder von Verzeichnis zu Verzeichnis – gehangelt wird.
Um auf einen Ast in der nächstliegenden untergeordneten Ebene zuzugreifen, muss die Methode `get_child()` verwendet werden. Sie gibt eine Referenz auf ein Objekt vom gleichen Typ zurück, für den `get_child()` aufgerufen wird – im Beispiel 25.1 eine Referenz auf `boost::property_tree::ptree`. Da jeder Ast seinerseits

Äste haben kann und es keine strukturellen Unterschiede zwischen Ästen auf höheren und niedrigeren Ebenen gibt, wird der gleiche Typ verwendet.
Nach drei Aufrufen von `get_child()` wird der `boost::property_tree::ptree` erhalten, der dem Verzeichnis System entspricht. Um den Wert zu lesen, der zu Beginn des Beispiels mit `put()` gespeichert wurde, wird `get_value()` aufgerufen.
Beachten Sie, dass es sich bei `get_value()` um eine Template-Funktion handelt und Sie den Typ des Rückgabewerts als Parameter übergeben müssen. So kann `get_value()` automatisch eine Typumwandlung vornehmen.

Beispiel 25.2 Datenzugriff mit `basic_ptree<std::string, int>`

```
#include <boost/property_tree/ptree.hpp>
#include <utility>
#include <iostream>

int main()
{
  typedef boost::property_tree::basic_ptree<std::string, int> ptree;
  ptree pt;
  pt.put(ptree::path_type{"C:\\Windows\\System", '\\'}, 20);
  pt.put(ptree::path_type{"C:\\Windows\\Cursors", '\\'}, 50);

  ptree &windows = pt.get_child(ptree::path_type{"C:\\Windows", '\\'});
  int files = 0;
  for (const std::pair<std::string, ptree> &p : windows)
    files += p.second.get_value<int>();
  std::cout << files << '\n';
}
```

Im Beispiel 25.2 wurden zwei Änderungen vorgenommen, um Pfade auf Verzeichnisse und die Anzahl der Dateien in Verzeichnissen einfacher speichern zu können. Zum einen werden Pfade wie unter Windows gewohnt mit einem Backslash als Trennzeichen an `put()` übergeben. Zum anderen wird die Anzahl der Dateien als int-Zahl gespeichert.
Boost.PropertyTree verwendet standardmäßig den Punkt als Trennzeichen für Schlüssel. Möchten Sie ein anderes Zeichen wie beispielsweise den Backslash verwenden, übergeben Sie den Schlüssel nicht als String an `put()`. Stattdessen packen Sie ihn in ein Objekt vom Typ `boost::property_tree::ptree::path_type`. Dem Konstruktor dieser Klasse, die von `boost::property_tree::ptree` abhängt, kann nicht nur der Schlüssel übergeben werden, sondern als zweiter Parameter das Zeichen, das als Trennzeichen verwendet werden soll. Auf diese Weise ist es im Beispiel 25.2 möglich, einen Pfad wie C:\Windows\System zu verwenden, ohne Backslashs durch Punkte ersetzen zu müssen.
Die von Boost.Property angebotene Datenstruktur `boost::property_tree::ptree` basiert auf einer Template-Klasse `boost::property_tree::basic_ptree`. Weil Schlüssel und Werte häufig Strings sind, ist `boost::property_tree::ptree` vordefiniert. Sie können aber auf `boost::property_tree::basic_ptree` zugreifen und andere Typen für Schlüssel und Werte verwenden. Die im Beispiel 25.2 definierte Datenstruktur verwendet int für Werte, um die Anzahl der Dateien in einem Verzeichnis nicht als String speichern zu müssen, sondern als Zahl.
`boost::property_tree::ptree` stellt die von Containern bekannten Methoden `begin()` und `end()` zur Verfügung. Eine Besonderheit ist jedoch, dass Sie bei `boost::property_tree::ptree` ausschließlich über Äste in einer Ebene iterieren. Im Beispiel 25.2 findet die Iteration über die Unterverzeichnisse von C:\Windows statt. Es ist nicht möglich, einen Iterator zu erhalten, der über sämtliche Äste in allen Ebenen iteriert.
In der `for`-Schleife im Beispiel 25.2 wird auf die Anzahl der Dateien in den Unterverzeichnissen von C:\Windows zugegriffen, um sie zu summieren. Als Ergebnis gibt das Programm `70` aus. Dabei wird nicht direkt auf Objekte vom Typ `ptree` zugegriffen. Stattdessen wird über Elemente vom Typ `std::pair<std::string, ptree>` iteriert. In **`first`** ist der Schlüssel des aktuellen Asts gespeichert. Das sind im Beispiel 25.2 System und Cursors. Über **`second`** wird auf das neuerliche Objekt vom Typ `ptree` zugegriffen, das mögliche Unterverzeichnisse enthält. Im Beispiel soll jedoch lediglich der Wert erhalten werden, der den Verzeichnissen System und Cursors zugeordnet ist. Dazu wird wie bereits im Beispiel 25.1 die Methode `get_value()` aufgerufen.
Die Datenstruktur `boost::property_tree::ptree` ist insofern bemerkenswert, als dass sie ausschließlich den Wert des aktuellen Asts speichert – nicht dessen Schlüsselnamen. Sie können über `get_value()` den Wert erhalten. Es gibt aber keine Methode, um den Schlüssel abzurufen. Dieser ist als Wert in der übergeordneten Datenstruktur `boost::property_tree::ptree` gespeichert, die die höhere Ebene repräsentiert. Das erklärt auch, warum in der `for`-Schleife auf Elemente vom Typ `std::pair<std::string, ptree>` zugegriffen wird.

Beispiel 25.3 Datenzugriff mit einem Translator

```
#include <boost/property_tree/ptree.hpp>
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>

struct string_to_int_translator
{
  typedef std::string internal_type;
  typedef int external_type;

  boost::optional<int> get_value(const std::string &s)
  {
    char *c;
    long l = std::strtol(s.c_str(), &c, 10);
    return boost::make_optional(c != s.c_str(), static_cast<int>(l));
  }
};

int main()
{
  typedef boost::property_tree::iptree ptree;
  ptree pt;
  pt.put(ptree::path_type{"C:\\Windows\\System", '\\'}, "20 files");
  pt.put(ptree::path_type{"C:\\Windows\\Cursors", '\\'}, "50 files");

  string_to_int_translator tr;
  int files =
    pt.get<int>(ptree::path_type{"c:\\windows\\system", '\\'}, tr) +
    pt.get<int>(ptree::path_type{"c:\\windows\\cursors", '\\'}, tr);
  std::cout << files << '\n';
}
```

Im Beispiel 25.3 kommt mit `boost::property_tree::iptree` eine weitere von Boost.PropertyTree vordefinierte Datenstruktur zum Einsatz. Sie verhält sich grundsätzlich genauso wie `boost::property_tree::ptree`. Der einzige Unterschied ist, dass bei Schlüsseln nicht zwischen Groß- und Kleinschreibung unterschieden wird. So kann zum Beispiel ein Wert, der mit dem Schlüssel C:\Windows\System gespeichert wurde, über c:\windows\system gelesen werden.

Im Gegensatz zum Beispiel 25.1 wird nicht mehr nacheinander über `get_child()` auf Unterebenen zugegriffen. So wie mit `put()` ein Wert direkt geschrieben werden kann, kann mit `get()` ein Wert direkt gelesen werden. Die Angabe des Schlüssels erfolgt auf die gleiche Weise – also zum Beispiel auch über `boost::property_tree::iptree::path_type`.

Wie bei `get_value()` handelt es sich auch bei `get()` um eine Template-Funktion. Sie müssen den Typ des Rückgabewerts als Parameter angeben. Boost.PropertyTree führt automatisch eine Typkonvertierung durch. Boost.PropertyTree verwendet für Typkonvertierungen *Translatoren*. Die Bibliothek stellt einige Translatoren zur Verfügung. Diese basieren auf Streams und können Typumwandlungen automatisch vornehmen.

Im Beispiel 25.3 wird ein Translator `string_to_int_translator` definiert, der einen Wert vom Typ `std::string` in ein int umwandelt. Der Translator wird als zusätzlicher Parameter an `get()` übergeben. Da er ausschließlich zum Lesen verwendet wird, besitzt er lediglich eine Methode `get_value()`. Würde er auch zum Schreiben verwendet werden – er würde dann als zusätzlicher Parameter an `put()` übergeben werden – müsste auch eine Methode `put_value()` definiert werden.

`get_value()` bekommt den Wert mit dem Typ übergeben, wie er in **`pt`** verwendet wird. Der Rückgabewert besitzt jedoch nicht allein den Typ, in den umgewandelt werden soll. Wie Sie sehen, wird auf `boost::optional` zugegriffen. Diese Klasse wird verwendet, weil eine Typumwandlung nicht immer gelingen muss. Würde im Beispiel 25.3 ein Wert gespeichert, der nicht mit `std::strtol()` in ein int umgewandelt werden könnte, müsste ein Objekt vom Typ `boost::optional` zurückgegeben werden, das leer ist.

Beachten Sie, dass ein Translator darüber hinaus zwei Typen internal_type und external_type definieren muss. Wenn Sie außerdem `put_value()` definieren möchten, um eine Typumwandlung auch beim Speichern von Daten vornehmen zu können, muss diese Methode ähnlich wie `get_value()` definiert werden.

Wenn Sie Beispiel 25.3 dahingehend ändern, dass zum Beispiel anstatt dem Wert „20 files“ lediglich „20“ gespeichert wird, können Sie `get_value()` aufrufen, ohne einen Translator übergeben zu müssen. Die von Boost.PropertyTree definierten Translatoren können eine Typumwandlung von `std::string` zu int durchführen. Die Typumwandlung wird jedoch nur dann als erfolgreich abgeschlossen betrachtet, wenn der gesamte String umgewandelt werden konnte. Der String darf also keine Buchstaben enthalten, damit die mit Boost.PropertyTree ausgelieferten Translatoren verwendet werden können. Da `std::strtol()` eine Typumwandlung durchführen kann, solange ein String mit Ziffern beginnt, ist der Translator `string_to_int_translator`, der im Beispiel 25.3 zum Einsatz kommt, liberaler.

Beispiel 25.4 Verschiedene Methoden von `boost::property_tree::ptree`

```
#include <boost/property_tree/ptree.hpp>
#include <utility>
#include <iostream>

using boost::property_tree::ptree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");

  boost::optional<std::string> c = pt.get_optional<std::string>("C:");
  std::cout << std::boolalpha << c.is_initialized() << '\n';

  pt.put_child("D:.Program Files", ptree{"50 files"});
  pt.add_child("D:.Program Files", ptree{"60 files"});

  ptree d = pt.get_child("D:");
  for (const std::pair<std::string, ptree> &p : d)
    std::cout << p.second.get_value<std::string>() << '\n';

  boost::optional<ptree&> e = pt.get_child_optional("E:");
  std::cout << e.is_initialized() << '\n';
}
```

Sie können die Methode `get_optional()` verwenden, wenn Sie den Wert eines Schlüssels lesen wollen, aber nicht sicher sind, ob der Schlüssel existiert. `get_optional()` gibt den Wert in einem Objekt vom Typ `boost::optional` zurück, das leer ist, wenn der Schlüssel nicht gefunden wurde. Ansonsten funktioniert `get_optional()` wie `get()`.
Die beiden Methoden `put_child()` und `add_child()` machen auf den ersten Blick das Gleiche wie `put()`. Der Unterschied ist, dass `put()` lediglich ein Schlüssel/Wert-Paar erstellt, während Sie mit `put_child()` und `add_child()` einen kompletten Baum in einen anderen hängen. Achten Sie auf den zweiten Parameter, der an `put_child()` und `add_child()` übergeben wird: Es handelt sich um ein Objekt vom Typ `boost::property_tree::ptree`.
Der Unterschied zwischen `put_child()` und `add_child()` ist, dass `put_child()` auf einen existierenden Schlüssel zugreift, sollte er bereits existieren. `add_child()` hingegen fügt dem Baum immer einen neuen Schlüssel hinzu. So hat der Baum im Beispiel 25.4 tatsächlich zwei Schlüssel „D:.Program Files“. Das kann je nach Anwendungsfall verwirrend sein. Stellt der Baum ein Verzeichnissystem dar, sollte es idealerweise nicht zwei gleichlautende Pfade geben. Da `boost::property_tree::ptree` gleichnamige Schlüssel erlaubt, müssen Sie in diesem Fall selbst darauf achten, nicht mehrere gleichnamige Schlüssel zu erstellen.
Wenn Sie Beispiel 25.4 ausführen, werden Ihnen in der `for`-Schleife die Werte ausgegeben, die den Schlüsseln unterhalb von „D:“ zugeordnet sind. Das Beispiel gibt `50 files` und `60 files` aus, was beweist, dass es tatsächlich zwei gleichnamige Schlüssel „D:.Program Files“ gibt.
Als letzte Methode wird Ihnen im Beispiel 25.4 `get_child_optional()` vorgestellt. Grundsätzlich verwenden Sie diese Methode wie `get_child()`. Diese Methode gibt ein Objekt vom Typ `boost::optional` zurück. Sie können `get_child_optional()` aufrufen, wenn Sie nicht sicher sind, ob ein Schlüssel existiert.

Beispiel 25.5 `boost::property_tree::ptree` im JSON-Format speichern

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>
#include <iostream>
```

```
using namespace boost::property_tree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");
  pt.put("C:.Windows.Cursors", "50 files");

  json_parser::write_json("file.json", pt);

  ptree pt2;
  json_parser::read_json("file.json", pt2);

  std::cout << std::boolalpha << (pt == pt2) << '\n';
}
```

Boost.PropertyTree bietet nicht nur einen Container an, um Daten im Speicher zu verwalten. Die Bibliothek stellt wie im Beispiel 25.5 zu sehen auch Funktionen zur Verfügung, um einen `boost::property_tree::ptree` in einer Datei zu speichern und von einer Datei zu laden.
Über die Headerdatei `boost/property_tree/json_parser.hpp` erhalten Sie Zugriff auf die Funktionen `boost::property_tree::json_parser::write_json()` und `boost::property_tree::json_parser::read_json()`. Diese ermöglichen es Ihnen, einen `boost::property_tree::ptree` im JSON-Format zu speichern und zu laden. So können Sie Konfigurationsdateien im JSON-Format unterstützen.
Wenn Sie auf Funktionen zugreifen möchten, die einen `boost::property_tree::ptree` in einer Datei speichern oder von einer Datei laden, müssen Sie Headerdateien wie `boost/property_tree/json_parser.hpp` explizit einbinden. Es reicht nicht aus, lediglich auf `boost/property_tree/ptree.hpp` zuzugreifen.
Neben `boost::property_tree::json_parser::write_json()` und `boost::property_tree::json_parser::read_json()` stellt Boost.PropertyTree Funktionen für weitere Datenformate zur Verfügung. Verwenden Sie `boost::property_tree::ini_parser::write_ini()` und `boost::property_tree::ini_parser::read_ini()` aus `boost/property_tree/ini_parser.hpp`, um INI-Dateien zu unterstützen. Mit `boost::property_tree::xml_parser::write_xml()` und `boost::property_tree::xml_parser::read_xml()` aus `boost/property_tree/xml_parser.hpp` können Sie Daten im XML-Format speichern und laden. Und mit `boost::property_tree::info_parser::write_info()` und `boost::property_tree::info_parser::read_info()` aus `boost/property_tree/info_parser.hpp` wird ein Format unterstützt, das explizit für Boost.PropertyTree entwickelt wurde und zur Serialisierung von Baumstrukturen optimiert ist.
Keines der vier unterstützten Formate garantiert, dass ein `boost::property_tree::ptree` nach dem Speichern und Laden genauso aussieht wie zuvor. Im JSON-Format kann zum Beispiel eine Typinformation verloren gehen, weil `boost::property_tree::ptree` nicht zwischen `true` und „true“ unterscheiden kann. Der Typ des Werts ist immer derselbe. Auch wenn die vorgestellten Funktionen es sehr einfach machen, einen `boost::property_tree::ptree` zu speichern und zu laden, sollten Sie daran denken, dass Boost.PropertyTree die Formate nicht vollständig unterstützt. Das Augenmerk dieser Bibliothek liegt auf dem Container `boost::property_tree::ptree` und nicht auf der Unterstützung verschiedener Datenformate.

Kapitel 26

Boost.DynamicBitset

Die Bibliothek Boost.DynamicBitset bietet mit `boost::dynamic_bitset` eine Klasse an, die grundsätzlich genauso verwendet wird wie `std::bitset`. Der entscheidende Unterschied ist, dass die Anzahl der Bits bei `std::bitset` zur Kompilierung angegeben werden muss, während dies bei `boost::dynamic_bitset` zur Laufzeit geschieht.
Um `boost::dynamic_bitset` einsetzen zu können, müssen Sie die Headerdatei `boost/dynamic_bitset.hpp` einbinden.

Beispiel 26.1 `boost::dynamic_bitset` in Aktion

```
#include <boost/dynamic_bitset.hpp>
#include <iostream>

int main()
{
  boost::dynamic_bitset<> db{3, 4};

  db.push_back(true);

  std::cout.setf(std::ios::boolalpha);
  std::cout << db.size() << '\n';
  std::cout << db.count() << '\n';
  std::cout << db.any() << '\n';
  std::cout << db.none() << '\n';

  std::cout << db[0].flip() << '\n';
  std::cout << ~db[3] << '\n';
  std::cout << db << '\n';
}
```

`boost::dynamic_bitset` ist ein Template, dem Sie bei der Instanziierung keine Parameter übergeben müssen – es werden dann Standardwerte verwendet. Wichtiger sind die Parameter, die Sie dem Konstruktor übergeben. So wird im Beispiel 26.1 ein Konstruktor aufgerufen, der **db** so initialisiert, dass 3 Bits zur Verfügung stehen. Der zweite Parameter, der dem Konstruktor übergeben wird, initialisiert die Bits. Die Zahl 4 bedeutet, dass von den 3 Bits nur das höchstwertige Bit gesetzt ist – also das Bit ganz links.
Sie können die Anzahl der Bits in einem Objekt vom Typ `boost::dynamic_bitset` jederzeit ändern. So steht wie im Beispiel 26.1 zu sehen eine Methode `push_back()` zur Verfügung, mit der der Datenstruktur ein Bit hinzugefügt wird. Dieses Bit wird das neue höchstwertige Bit. Der Aufruf von `push_back()` im Beispiel führt dazu, dass **db** vier Bits besitzt, von denen die beiden höchstwertigen Bits gesetzt sind. **db** speichert die Zahl 12.
Beachten Sie, dass es auch möglich ist, die Anzahl der Bits zu verringern. Die entsprechende Methode, die Sie hierzu aufrufen müssen, heißt `resize()`. Je nachdem, welchen Parameter Sie an `resize()` übergeben, werden Bits hinzugefügt oder entfernt.
`boost::dynamic_bitset` stellt Methoden zur Verfügung, um Daten über die gesamte Datenstruktur abzufragen oder um auf einzelne Bits in der Datenstruktur zuzugreifen. So gibt `size()` die Anzahl der Bits zurück und `count()` die Anzahl der gesetzten Bits. Während `any()` `true` zurückgibt, wenn mindestens ein Bit gesetzt ist, gibt `none()` `true` zurück, wenn kein Bit gesetzt ist.
Um auf einzelne Bits zuzugreifen, verwenden Sie die von Arrays bekannte Syntax. Sie erhalten dann eine Referenz auf eine interne Klasse, die das entsprechende Bit repräsentiert und Methoden anbietet, um das Bit zu bear-

beiten. So können Sie zum Beispiel `flip()` aufrufen, um ein Bit umzudrehen. Die aus C++ bekannten bitweisen Operatoren wie `operator~` stehen ebenfalls zur Verfügung. Letztendlich bietet `boost::dynamic_bitset` all das, was von `std::bitset` bekannt ist.
Beachten Sie, dass `boost::dynamic_bitset` keine Iteratoren anbietet. Auch hier verhält sich `boost::dynamic_bitset` wie `std::bitset`.

Kapitel 27

Boost.Tribool

Die Bibliothek Boost.Tribool bietet mit `boost::logic::tribool` einen Typ an, der bool ähnelt. Während bool zwischen zwei Status unterscheidet, kennt `boost::logic::tribool` jedoch drei.
Um `boost::logic::tribool` verwenden zu können, müssen Sie die Headerdatei `boost/logic/tribool.hpp` einbinden.

Beispiel 27.1 Drei Status von `boost::logic::tribool`

```
#include <boost/logic/tribool.hpp>
#include <iostream>

using namespace boost::logic;

int main()
{
  tribool b;
  std::cout << std::boolalpha << b << '\n';

  b = true;
  b = false;
  b = indeterminate;
  if (b)
    ;
  else if (!b)
    ;
  else
    std::cout << "indeterminate\n";
}
```

Eine Variable vom Typ `boost::logic::tribool` kann auf `true`, `false` und `indeterminate` gesetzt werden. Der Standardkonstruktur initialisiert die Variable mit `false`. So gibt Beispiel 27.1 zuerst `false` auf die Standardausgabe aus.
Die `if`-Kontrollstruktur im Beispiel 27.1 zeigt Ihnen, wie **b** richtig abgefragt wird. Sie müssen explizit auf `true` und `false` überprüfen. Ist wie im Beispiel die Variable auf `indeterminate` gesetzt, wird der `else`-Block ausgeführt.
Boost.Tribool bietet auch eine Funktion `boost::logic::indeterminate()` an. Wenn Sie dieser Funktion eine Variable vom Typ `boost::logic::tribool` übergeben, die auf `indeterminate` gesetzt ist, wird `true` zurückgegeben. Ist die Variable auf `true` oder `false` gesetzt, wird `false` zurückgegeben.

Beispiel 27.2 Logische Operatoren mit `boost::logic::tribool`

```
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>
#include <iostream>

using namespace boost::logic;

int main()
{
  std::cout.setf(std::ios::boolalpha);
```

```
  tribool b1 = true;
  std::cout << (b1 || indeterminate) << '\n';
  std::cout << (b1 && indeterminate) << '\n';

  tribool b2 = false;
  std::cout << (b2 || indeterminate) << '\n';
  std::cout << (b2 && indeterminate) << '\n';

  tribool b3 = indeterminate;
  std::cout << (b3 || b3) << '\n';
  std::cout << (b3 && b3) << '\n';
}
```

So wie auf Variablen vom Typ bool können auch auf Variablen vom Typ `boost::logic::tribool` logische Operatoren angewandt werden. Genaugenommen ist dies die einzige Möglichkeit, mit Variablen vom Typ `boost::logic::tribool` zu arbeiten. Die Klasse bietet keine Methoden an.
Wenn Sie Beispiel 27.2 ausführen, sehen Sie, dass für `b1 || indeterminate` `true`, für `b2 && indeterminate` `false` und für alle anderen Fälle `indeterminate` zurückgegeben wird. Wenn Sie sich die Operationen und ihre Ergebnisse im Detail ansehen, stellen Sie fest, dass sich `boost::logic::tribool` so verhält, wie man es intuitiv erwarten würde. Sie finden in der Dokumentation von Boost.Tribool jedoch auch Tabellen, die anzeigen, welche Verknüpfungen mit welchen Operatoren zu welchen Ergebnissen führen.
Beispiel 27.2 zeigt auch, wie für Variablen vom Typ `boost::logic::tribool` `true`, `false` oder `indeterminate` auf die Standardausgabe ausgegeben wird. Dazu muss die Headerdatei `boost/logic/tribool_io.hpp` eingebunden und das Flag **`std::ios::boolalpha`** für die Standardausgabe gesetzt werden.
Boost.Tribool bietet außerdem ein Makro `BOOST_TRIBOOL_THIRD_STATE` an, mit dem ein anderer Wert als `indeterminate` verwendet werden kann. So könnte zum Beispiel `dontknow` anstatt `indeterminate` geschrieben werden.

Kapitel 28

Boost.CompressedPair

Boost.CompressedPair stellt mit `boost::compressed_pair` eine Klasse zur Verfügung, die sich wie `std::pair` verhält. Ist einer oder sind beide Template-Parameter leere Klassen, benötigt `boost::compressed_pair` jedoch weniger Speicherplatz. Hierbei kommt eine Optimierung zum Einsatz, die auf Englisch empty base class optimization heißt.
Um `boost::compressed_pair` verwenden zu können, müssen Sie die Headerdatei `boost/compressed_pair.hpp` einbinden.

Beispiel 28.1 Weniger Speicherbedarf bei `boost::compressed_pair`

```
#include <boost/compressed_pair.hpp>
#include <utility>
#include <iostream>

struct empty {};

int main()
{
  std::pair<int, empty> p;
  std::cout << sizeof(p) << '\n';

  boost::compressed_pair<int, empty> cp;
  std::cout << sizeof(cp) << '\n';
}
```

Wenn Sie Beispiel 28.1 ausführen, wird Ihnen für **cp** eine geringere Speichergröße ausgegeben als für **p**. Mit Visual C++ 2013 erstellt und auf einem 64-Bit Windows 7-System ausgeführt gibt das Beispielprogramm für `sizeof(cp)` `4` und für `sizeof(p)` `8` aus.
Beachten Sie, dass es einen weiteren großen Unterschied zwischen `std::pair` und `boost::compressed_pair` gibt: Während Sie bei `std::pair` über zwei Eigenschaften **`first`** und **`second`** auf gespeicherte Werte zugreifen, rufen Sie bei `boost::compressed_pair` zwei Methoden `first()` und `second()` auf.

Teil V

Algorithmen

Die Boost-Bibliotheken bieten zahlreiche Algorithmen an, die die Algorithmen aus der Standardbibliothek ergänzen.

- Die wichtigste Bibliothek ist Boost.Algorithm, in der nüztliche Algorithmen gesammelt und zur Verfügung gestellt werden.
- Boost.Range stellt ebenfalls Algorithmen zur Verfügung, definiert mit der Range jedoch ein neues Konzept, das den Umgang mit Algorithmen vereinfachen soll.
- Boost.Graph wiederum ist auf Graphen spezialisiert und bietet zum Beispiel Algorithmen an, um die kürzeste Strecke zwischen zwei Punkten zu finden.

Einige Algorithmus-Bibliotheken werden in anderen Teilen des Buchs vorgestellt. So finden Sie zum Beispiel Algorithmen für Strings in der Bibliothek Boost.StringAlgorithms, die in Teil II vorgestellt wird.

Kapitel 29

Boost.Algorithm

Boost.Algorithm stellt Algorithmen zur Verfügung, die die Algorithmen aus der Standardbibliothek ergänzen. Dabei versucht Boost.Algorithm nicht, neue Konzepte einzuführen, wie es zum Beispiel Boost.Range macht. Die von Boost.Algorithm angebotenen Algorithmen ähneln denen aus der Standardbibliothek und erwarten zum Beispiel ebenfalls üblicherweise mindestens zwei Iteratoren.
Beachten Sie, dass es zahlreiche Algorithmen in anderen Boost-Bibliotheken gibt. So finden Sie beispielsweise Algorithmen zum Bearbeiten von Strings in Boost.StringAlgorithms. Die von Boost.Algorithm angebotenen Algorithmen sind nicht an bestimmte Klassen wie beispielsweise `std::string` gebunden, sondern können wie die Algorithmen aus der Standardbibliothek mit beliebigen Containern verwendet werden.

Beispiel 29.1 Mit `boost::algorithm::one_of_equal()` auf genau einen Wert überprüfen

```
#include <boost/algorithm/cxx11/one_of.hpp>
#include <array>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::array<int, 6> a{{0, 5, 2, 1, 4, 3}};
  auto predicate = [](int i){ return i == 4; };
  std::cout.setf(std::ios::boolalpha);
  std::cout << one_of(a.begin(), a.end(), predicate) << '\n';
  std::cout << one_of_equal(a.begin(), a.end(), 4) << '\n';
}
```

Boost.Algorithm stellt mit `boost::algorithm::one_of()` einen Algorithmus zur Verfügung, um zu überprüfen, ob eine Bedingung genau einmal wahr ist. Die zu überprüfende Bedingung wird dabei als Prädikat an die Funktion übergeben. So gibt der Aufruf von `boost::algorithm::one_of()` im Beispiel 29.1 `true` zurück, weil sich die Zahl 4 genau einmal in **a** befindet.
Wenn Sie Werte in einem Container auf Gleichheit überprüfen möchten, können Sie auf die Funktion `boost::algorithm::one_of_equal()` zugreifen. Sie übergeben dann kein Prädikat, sondern den auf Gleichheit zu überprüfenden Wert. So gibt der Aufruf von `boost::algorithm::one_of_equal()` im Beispiel 29.1 ebenfalls `true` zurück.
`boost::algorithm::one_of()` ergänzt die Algorithmen `std::all_of()`, `std::any_of()` und `std::none_of()` aus der Standardbibliothek. Da diese Algorithmen erst mit C++11 in die Standardbibliothek aufgenommen wurden, stellt Boost.Algorithm mit `boost::algorithm::all_of()`, `boost::algorithm::any_of()` und `boost::algorithm::none_of()` die Algorithmen auch Entwicklern zur Verfügung, die nicht mit C++11 arbeiten. Sie finden diese Algorithmen in den Headerdateien `boost/algorithm/cxx11/all_of.hpp`, `boost/algorithm/cxx11/any_of.hpp` und `boost/algorithm/cxx11/none_of.hpp`.
Neben den Algorithmen `boost::algorithm::all_of()`, `boost::algorithm::any_of()` und `boost::algorithm::none_of()` bietet Boost.Algorithm auch `boost::algorithm::all_of_equal()`, `boost::algorithm::any_of_equal()` und `boost::algorithm::none_of_equal()` an.
Boost.Algorithm bietet weitere Algorithmen aus der C++11-Standardbibliothek an. So können Sie zum Beispiel auf `boost::algorithm::is_partitioned()`, `boost::algorithm::is_permutation()`, `boost::algorithm::copy_n()`, `boost::algorithm::find_if_not()` oder `boost::algorithm::iota()` zu-

greifen. Diese Funktionen verhalten sich genauso wie die Funktionen aus der C++11-Standardbibliothek und richten sich vor allem an Entwickler, die nicht mit C++11 arbeiten. Boost.Algorithm bietet jedoch einige Varianten der Funktionen an, die auch für C++11-Entwickler interessant sein können.

Beispiel 29.2 Mehr Varianten von C++11-Algorithmen von Boost.Algorithm

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/algorithm/cxx11/is_sorted.hpp>
#include <boost/algorithm/cxx11/copy_if.hpp>
#include <vector>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v;
  iota_n(std::back_inserter(v), 10, 5);
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_increasing(v) << '\n';
  std::ostream_iterator<int> out{std::cout, ","};
  copy_until(v, out, [](int i){ return i > 12; });
}
```

Boost.Algorithm bietet in der Headerdatei `boost/algorithm/cxx11/iota.hpp` den C++11-Algorithmus `boost::algorithm::iota()` an, der aufsteigende Zahlen generiert. Dazu müssen dieser Funktion zwei Iteratoren auf den Anfang und das Ende eines Containers übergeben werden. Die Elemente im Container werden daraufhin mit aufsteigenden Zahlen überschrieben.
Im Beispiel 29.2 kommt nicht `boost::algorithm::iota()`, sondern `boost::algorithm::iota_n()` zum Einsatz. Diese Funktion erwartet lediglich einen Iterator, auf den zu generierende Zahlen ausgegeben werden sollen. Die Anzahl der zu generierenden Zahlen wird als dritter Parameter an `boost::algorithm::iota_n()` übergeben.
`boost::algorithm::is_increasing()` ist in der Headerdatei `boost/algorithm/cxx11/is_sorted.hpp` definiert, in der Sie auch den C++11-Algorithmus `boost::algorithm::is_sorted()` finden. `boost::algorithm::is_increasing()` ist gleichbedeutend mit `boost::algorithm::is_sorted()`. Der Funktionsname drückt jedoch klar aus, dass auf eine aufsteigende Sortierung überprüft wird. Neben `boost::algorithm::is_increasing()` definiert die Headerdatei auch eine Funktion `boost::algorithm::is_decreasing()`.
Beachten Sie, dass im Beispiel 29.2 **v** direkt an `boost::algorithm::is_increasing()` übergeben wird. Die von Boost.Algorithm angebotenen Funktionen liegen auch in einer range-basierten Variante vor, so dass Sie Container direkt übergeben können und nicht wie bei Algorithmen aus der Standardbibliothek zwei Iteratoren einzeln übergeben müssen.
`boost::algorithm::copy_until()` finden Sie in der Headerdatei `boost/algorithm/cxx11/copy_if.hpp`. Es handelt sich hierbei um eine weitere Variante von `std::copy()`. So steht neben `boost::algorithm::copy_until()` auch die Funktion `boost::algorithm::copy_while()` zur Verfügung.
Wenn Sie Beispiel 29.2 ausführen, wird `true` als Ergebnis von `boost::algorithm::is_increasing()` ausgegeben. Außerdem werden mit `boost::algorithm::copy_until()` die Zahlen 10, 11 und 12 in die Standardausgabe geschrieben.

Beispiel 29.3 C++14-Algorithmen von Boost.Algorithm

```
#include <boost/algorithm/cxx14/equal.hpp>
#include <boost/algorithm/cxx14/mismatch.hpp>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v{1, 2};
  std::vector<int> w{1, 2, 3};
  std::cout.setf(std::ios::boolalpha);
```

```
  std::cout << equal(v.begin(), v.end(), w.begin(), w.end()) << '\n';
  auto pair = mismatch(v.begin(), v.end(), w.begin(), w.end());
  if (pair.first != v.end())
    std::cout << *pair.first << '\n';
  if (pair.second != w.end())
    std::cout << *pair.second << '\n';
}
```

Boost.Algorithm bietet nicht nur Algorithmen aus der C++11-Standardbibliothek an. Es stehen auch Algorithmen zur Verfügung, die mit C++14 in die Standardbibliothek aufgenommen wurden. So greift Beispiel 29.3 auf zwei Funktionen `boost::algorithm::equal()` und `boost::algorithm::mismatch()` zu, die erst seit C++14 zur Standardbibliothek gehören. Im Gegensatz zu den gleichnamigen Funktionen, die seit C++98 Teil der Standardbibliothek sind, werden den neuen Funktionen vier und nicht drei Iteratoren übergeben. Die Algorithmen im Beispiel 29.3 erwarten nicht, dass die zweite Sequenz mindestens so viele Elemente wie die erste Sequenz besitzt.
Während `boost::algorithm::equal()` einen Rückgabewert vom Typ bool hat, gibt `boost::algorithm::mismatch()` zwei Iteratoren in einem `std::pair` zurück. **`first`** und **`second`** zeigen auf die beiden Elemente in der ersten und zweiten Sequenz, die die ersten sind, die sich unterscheiden. Beachten Sie, dass die entsprechenden Iteratoren auch auf das Ende einer Sequenz zeigen können.
Wenn Sie Beispiel 29.3 ausführen, wird `false` und `3` ausgegeben. `false` ist der Rückgabewert von `boost::algorithm::equal()`, 3 das dritte Elemente in **`w`**. Da die ersten beiden Elemente in **`v`** und **`w`** gleich sind, gibt `boost::algorithm::mismatch()` in **`first`** einen Iterator auf das Ende von **`v`** und in **`second`** einen Iterator auf das dritte Element in **`w`** zurück. Da **`first`** auf das Ende von **`v`** zeigt, wird der Iterator nicht dereferenziert, und es findet keine Ausgabe statt.

Beispiel 29.4 `boost::algorithm::hex()` und `boost::algorithm::unhex()`

```
#include <boost/algorithm/hex.hpp>
#include <vector>
#include <string>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<char> v{'C', '+', '+'};
  hex(v, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string s = "C++";
  std::cout << hex(s) << '\n';

  std::vector<char> w{'4', '3', '2', 'b', '2', 'b'};
  unhex(w, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string t = "432b2b";
  std::cout << unhex(t) << '\n';
}
```

Im Beispiel 29.4 kommen die beiden Funktionen `boost::algorithm::hex()` und `boost::algorithm::unhex()` zum Einsatz. Sie sind den gleichnamigen Funktionen aus der Datenbank MySQL nachempfunden und wandeln Zeichen in Hexadezimalwerte oder Hexadezimalwerte in Zeichen um.
Im Beispiel wird der Vektor **`v`** mit den drei Zeichen „C“, „+“ und „+“ an `boost::algorithm::hex()` übergeben. Die Funktion erwartet als zweiten Parameter einen Iterator, um die Hexadezimalzahlen ausgeben zu können. Wenn Sie das Programm ausführen, wird für „C“ `43` und für „+“ zweimal `2B` ausgegeben. Beim zweiten Aufruf von `boost::algorithm::hex()` geschieht das Gleiche, nur dass „C++“ als String übergeben und „432B2B“ als String zurückgegeben wird.
`boost::algorithm::unhex()` funktioniert ähnlich wie `boost::algorithm::hex()`, macht aber das Gegenteil. Wenn wie im Beispiel das Array **`w`** mit sechs Hexadezimalzeichen übergeben wird, werden jeweils zwei

Zeichen als ASCII-Code interpretiert. Das Gleiche geschieht beim zweiten Aufruf von `boost::algorithm::unhex()`, dem sechs Hexadezimalzeichen in einem String übergeben werden. In beiden Fällen wird `C++` in die Standardausgabe geschrieben.
Boost.Algorithm bietet weitere Algorithmen an. So stehen mehrere String-Matching-Algorithmen zur Verfügung, um effizient in Texten zu suchen. Die Dokumentation enthält eine Referenz mit einer Übersicht über alle verfügbaren Algorithmen.

Kapitel 30

Boost.Range

Boost.Range ist eine Bibliothek, die auf den ersten Blick ähnliche Algorithmen wie die Standardbibliothek bietet. So treffen Sie beispielsweise auf eine Funktion `boost::copy()`, die das Gleiche macht wie `std::copy()`. Der entscheidende Unterschied ist, dass `std::copy()` zwei Iteratoren erwartet, während Sie `boost::copy()` eine Range übergeben.

30.1 Algorithmen

Sie können sich eine *Range* als zwei Iteratoren vorstellen, die auf den Anfang und das Ende einer Gruppe an Elementen zeigen, über die iteriert werden kann. Da alle Container Iteratoren unterstützen, kann jeder Container als Range dargestellt werden. Da alle Algorithmen von Boost.Range als ersten Parameter eine Range erwarten, können Sie einen Container wie `std::vector` direkt übergeben. Sie müssen also nicht erst `begin()` und `end()` aufrufen, um zwei Iteratoren einzeln zu übergeben. So sind Sie davor geschützt, versehentlich den Start- und End-Iterator in der falschen Reihenfolge oder Iteratoren zu übergeben, die zu zwei verschiedenen Containern gehören.

Beispiel 30.1 Zählen mit `boost::count()`

```
#include <boost/range/algorithm.hpp>
#include <array>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 0, 1, 0, 1}};
  std::cout << boost::count(a, 0) << '\n';
}
```

Im Beispiel 30.1 wird der Algorithmus `boost::count()` verwendet. Dazu wurde die Headerdatei `boost/range/algorithm.hpp` eingebunden. Über diese Headerdatei erhalten Sie Zugriff auf alle Algorithmen, für die es ein Gegenstück in der Headerdatei `algorithm` in der Standardbibliothek gibt.
Wie bei allen von Boost.Range angebotenen Algorithmen muss als erster Parameter eine Range übergeben werden. Da Container auch Ranges sind, kann ein Objekt vom Typ `std::array` direkt an `boost::count()` übergeben werden. Da `boost::count()` das Gleiche macht wie `std::count()`, muss außerdem ein Wert übergeben werden, mit dem die Elemente in der Range verglichen werden sollen.
Wenn Sie Beispiel 30.1 ausführen, wird `3` auf die Standardausgabe ausgegeben, da **a** drei Nullen enthält.
Beispiel 30.2 stellt weitere Algorithmen vor, die wie `boost::count()` Algorithmen aus der Standardbibliothek ähneln.

Beispiel 30.2 Mit der Standardbibliothek verwandte Range-Algorithmen

```
#include <boost/range/algorithm.hpp>
#include <boost/range/numeric.hpp>
#include <array>
#include <iterator>
#include <iostream>

int main()
```

```
{
  std::array<int, 6> a{{0, 1, 2, 3, 4, 5}};
  boost::random_shuffle(a);
  boost::copy(a, std::ostream_iterator<int>{std::cout, ","});
  std::cout << "\n" << *boost::max_element(a) << '\n';
  std::cout << boost::accumulate(a, 0) << '\n';
}
```

`boost::random_shuffle()` macht das Gleiche wie `std::random_shuffle()` und ändert die Reihenfolge der Elemente in der Range nach dem Zufallsprinzip. Im Beispiel 30.2 verwendet `boost::random_shuffle()` einen Standardzufallsgenerator. Sie können der Funktion jedoch auch als zweiten Parameter einen Zufallsgenerator übergeben, wie sie in C++11 in der Headerdatei `random` oder von Boost.Random zur Verfügung gestellt werden.
`boost::copy()` macht das Gleiche wie `std::copy()`. Auch `boost::max_element()` und `boost::accumulate()` unterscheiden sich nicht von den gleichnamigen Algorithmen aus der Standardbibliothek. `boost::max_element()` gibt so wie `std::max_element()` einen Iterator auf das Element mit der größten Zahl zurück. Beachten Sie, dass für `boost::accumulate()` die Headerdatei `boost/range/numeric.hpp` eingebunden werden muss. So wie `std::accumulate()` in `numeric` definiert ist, ist auch `boost::accumulate()` in `boost/range/numeric.hpp` und nicht in `boost/range/algorithm.hpp` definiert.
Boost.Range bietet einige Algorithmen an, für die es kein Gegenstück in der Standardbibliothek gibt.

Beispiel 30.3 Range-Algorithmen ohne Gegenstück in der Standardbibliothek

```
#include <boost/range/algorithm_ext.hpp>
#include <array>
#include <deque>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 2, 3, 4, 5}};
  std::cout << std::boolalpha << boost::is_sorted(a) << '\n';
  std::deque<int> d;
  boost::push_back(d, a);
  boost::remove_erase(d, 2);
  boost::copy_n(d, 3, std::ostream_iterator<int>{std::cout, ","});
}
```

Um die im Beispiel 30.3 vorgestellten Algorithmen verwenden zu können, müssen Sie die Headerdatei `boost/range/algorithm_ext.hpp` einbinden. Über diese Headerdatei erhalten Sie Zugriff auf Algorithmen, für die es kein Gegenstück in der Standardbibliothek gibt.
`boost::is_sorted()` gibt an, ob die Werte in einer Range aufsteigend sortiert sind. Da dies im Beispiel 30.3 für **a** der Fall ist, gibt `boost::is_sorted()` `true` zurück. Sie können `boost::is_sorted()` auch ein Prädikat als zweiten Parameter übergeben, wenn Sie zum Beispiel auf absteigende Sortierung überprüfen wollen.
`boost::push_back()` erwartet als ersten Parameter einen Container und als zweiten Parameter eine Range. Der Container muss die Methode `push_back()` anbieten. Alle Werte in der Range werden über diese Methode dem Container hinzugefügt – in der Reihenfolge, in der sie in der Range vorliegen. Da **d** anfangs leer ist, speichert **d** nach dem Aufruf von `boost::push_back()` die gleichen Zahlen in der gleichen Reihenfolge wie **a**.
`boost::remove_erase()` entfernt die Zahl 2 aus **d**. Dieser Algorithmus ist insofern nützlich, als dass er einen Aufruf von `std::remove()` und der Methode `erase()` für den entsprechenden Container vereint. Sie müssen dank `boost::remove_erase()` nicht selbst zuerst den Iterator finden, der auf das entsprechende Element zeigt, um in einem zweiten Schritt den Iterator an `erase()` zu übergeben.
`boost::copy_n()` ist gleichbedeutend mit `boost::copy()`, erwartet als zweiten Parameter aber eine Angabe, wie viele Elemente aus der Range kopiert werden sollen. Im Beispiel 30.3 werden demnach lediglich die ersten drei Zahlen aus **d** auf die Standardausgabe ausgegeben. Da in der vorherigen Zeile die Zahl 2 aus **d** entfernt wurde, wird `0,1,3,` ausgegeben, wenn Sie das Programm ausführen.

30.2 Adapter

Die Standardbibliothek bietet einige Algorithmen an, denen ein Prädikat übergeben werden kann. So können Sie beispielsweise bei `std::count_if()` angeben, welche Elemente gezählt werden sollen. Boost.Range bietet mit `boost::count_if()` eine ähnliche Funktion. Dies geschieht jedoch ausschließlich der Vollständigkeit halber. Denn Boost.Range bietet neben Algorithmen Adapter an, die viele Algorithmen mit Prädikaten überflüssig machen.
Adapter können Sie sich als Filter vorstellen. Sie geben eine neue Range basierend auf einer anderen zurück. Dabei werden nicht zwangsläufig Daten kopiert. Da eine Range letztendlich ein Paar Iteratoren ist, gibt ein Adapter ein neues Paar zurück, mit deren Hilfe über die ursprüngliche Range iteriert wird, aber zum Beispiel bestimmte Elemente übersprungen werden. Wenn `boost::count()` mit einem derartigen Adapter verwendet wird, wird `boost::count_if()` nicht mehr benötigt. Algorithmen müssen nicht mehr mehrfach definiert werden, nur um ohne und mit Prädikat aufgerufen werden zu können.
Der Unterschied zwischen Algorithmen und Adaptern ist: Algorithmen iterieren über eine Range und verarbeiten Daten. Adapter hingegen geben eine neue Range zurück – neue Iteratoren. Es hängt von diesen neuen Iteratoren ab, wie über die Range iteriert wird. Es findet aber keine Iteration statt. Dazu müssen Sie erst einen Algorithmus aufrufen.

Beispiel 30.4 Eine Range mit `boost::adaptors::filter()` filtern

```
#include <boost/range/algorithm.hpp>
#include <boost/range/adaptors.hpp>
#include <array>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 5, 2, 1, 3, 4}};
  boost::copy(boost::adaptors::filter(a, [](int i){ return i > 2; }),
    std::ostream_iterator<int>{std::cout, ","});
}
```

Im Beispiel 30.4 kommt ein Adapter zum Einsatz, der Ranges filtern kann. Wie Sie sehen handelt es sich dabei schlichtweg um eine Funktion. So erwartet `boost::adaptors::filter()` als ersten Parameter die zu filternde Range und als zweiten Parameter ein Prädikat. Das Prädikat im Beispiel 30.4 entfernt alle Zahlen aus der Range, die nicht größer als 2 sind.
Beachten Sie, dass `boost::adaptors::filter()` die Range **a** nicht ändert. Eine Funktion wie `boost::adaptors::filter()` gibt eine neue Range zurück. Da eine Range letztendlich nichts anderes ist als ein Paar Iteratoren, verweist diese neue Range ebenfalls auf **a**. Die Iteratoren dieser neuen Range überspringen jedoch bei einer Iteratoren alle Zahlen, die nicht größer als 2 sind.
Führen Sie Beispiel 30.4 aus, wird Ihnen `5,3,4` auf die Standardausgabe ausgegeben.

Beispiel 30.5 `keys()`, `values()` und `indirect()` in Aktion

```
#include <boost/range/algorithm.hpp>
#include <boost/range/adaptors.hpp>
#include <array>
#include <map>
#include <string>
#include <utility>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 3> a{{0, 1, 2}};
  std::map<std::string, int*> m;
  m.insert(std::make_pair("a", &a[0]));
  m.insert(std::make_pair("b", &a[1]));
  m.insert(std::make_pair("c", &a[2]));

  boost::copy(boost::adaptors::keys(m),
    std::ostream_iterator<std::string>{std::cout, ","});
```

```
  boost::copy(boost::adaptors::indirect(boost::adaptors::values(m)),
    std::ostream_iterator<int>{std::cout, ","});
}
```

Beispiel 30.5 stellt Ihnen mit `boost::adaptors::keys()` und `boost::adaptors::values()` nicht nur zwei Adapter vor, um auf Schlüssel und Werte eines Containers wie vom Typ `std::map` zuzugreifen. Sie sehen auch, dass Sie Adapter verschachteln können. Da die Werte im Container `m` Zeiger sind, jedoch die Zahlen ausgegeben werden sollen, auf die sie zeigen, wird die von `boost::adaptors::values()` zurückgegebene Range an `boost::adaptors::indirect()` übergeben. Diesen Adapter können Sie immer dann verwenden, wenn die ursprüngliche Range aus Zeigern besteht, sie aber bei einer Iteration auf das zugreifen wollen, worauf die Zeiger verweisen. So gibt Beispiel 30.5 `a,b,c,0,1,2,` auf die Standardausgabe aus.

Beispiel 30.6 `boost::adaptors::tokenize()` – ein Adapter für Strings

```
#include <boost/range/algorithm.hpp>
#include <boost/range/adaptors.hpp>
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "The Boost C++ Libraries";
  boost::regex expr{"[\\w+]+"};
  boost::copy(boost::adaptors::tokenize(s, expr, 0,
    boost::regex_constants::match_default),
    std::ostream_iterator<std::string>{std::cout, ","});
}
```

Beispiel 30.6 stellt Ihnen einen Adapter für Strings vor. Sie können `boost::adaptors::tokenize()` verwenden, um eine Range auf einen String mit Hilfe eines regulären Ausdrucks zu erhalten. Sie übergeben dazu einen String und einen regulären Ausdruck vom Typ `boost::regex` an diese Funktion. Außerdem müssen Sie eine Zahl für eine Gruppe im regulären Ausdruck sowie ein Flag übergeben. Verwenden Sie keine Gruppe, übergeben Sie 0. Übergeben Sie als Flag **`boost::regex_constants::match_default`**, wenn Sie die Standardeinstellungen für reguläre Ausdrücke verwenden möchten. Sie können auch zum Beispiel **`boost::regex_constants::match_perl`** übergeben, wenn der reguläre Ausdruck gemäß den Regeln der Programmiersprache Perl angewandt werden soll.

30.3 Hilfsklassen und -funktionen

Die von Boost.Range angebotenen Algorithmen und Adapter basieren auf Templates. Sie müssen einen Container nicht explizit in eine Range umwandeln, um ihn an einen Algorithmus oder Adapter übergeben zu können. Boost.Range definiert jedoch auch einige Range-Klassen, von denen die wichtigste `boost::iterator_range` ist. `boost::iterator_range` wird benötigt, weil Adapter und einige Algorithmen Ranges zurückgeben und diese einen Typ haben müssen. Außerdem gibt es einige Hilfsfunktionen, um Ranges zu erstellen, deren Iteratoren alle für eine Iteration notwendigen Informationen besitzen. Diese Ranges zeigen über ihre Iteratoren auf keine Container oder andere Datenstrukturen. Auch hier kommt eine Klasse wie `boost::iterator_range` zum Einsatz.

Beispiel 30.7 Mit `boost::irange()` eine Range für Ganzzahlen erstellen

```
#include <boost/range/algorithm.hpp>
#include <boost/range/irange.hpp>
#include <iostream>

int main()
{
  boost::integer_range<int> ir = boost::irange(0, 3);
  boost::copy(ir, std::ostream_iterator<int>{std::cout, ","});
}
```

Im Beispiel 30.7 kommt die Funktion `boost::irange()` zum Einsatz. Diese Funktion erstellt eine Range für Ganzzahlen, ohne dass ein Container oder eine andere Datenstruktur verwendet werden muss. Sie übergeben der Funktion lediglich eine Unter- und Obergrenze, wobei die Obergrenze gerade nicht mehr zur Range gehört. `boost::irange()` gibt eine Range vom Typ `boost::integer_range` zurück. Diese Klasse ist von `boost::iterator_range` abgeleitet. `boost::iterator_range` ist ein Template, das als Parameter einen Iteratortyp erwartet. Der Iterator, der von `boost::irange()` verwendet wird, hängt eng mit dieser Funktion zusammen und ist ein Implementationsdetail. Um die Handhabung zu vereinfachen, steht `boost::integer_range` zur Verfügung. Diesem Template muss lediglich ein Typ für Ganzzahlen übergeben werden.
Wenn Sie Beispiel 30.7 ausführen, wird Ihnen `0,1,2` ausgegeben.

Beispiel 30.8 Mit `boost::istream_range()` eine Range für einen Inputstream erstellen

```
#include <boost/range/algorithm.hpp>
#include <boost/range/istream_range.hpp>
#include <iterator>
#include <iostream>

int main()
{
  boost::iterator_range<std::istream_iterator<int>> ir =
    boost::istream_range<int>(std::cin);
  boost::copy(ir, std::ostream_iterator<int>{std::cout, "\n"});
}
```

Beispiel 30.8 stellt Ihnen die Funktion `boost::istream_range()` vor, mit der Sie eine Range für einen Inputstream erstellen können. Diese Funktion gibt die Range als `boost::iterator_range` zurück. Das bedeutet, Sie müssen den Iteratortyp als Template-Parameter angeben.
Wenn Sie Beispiel 30.8 ausführen, eine Zahl eingeben und anschließend **Enter** drücken, wird die Zahl in der nächsten Zeile ausgegeben. Geben Sie erneut eine Zahl ein und drücken **Enter**, wird auch diese ausgegeben. Die Range, die `boost::istream_range()` zurückgibt, erlaubt es `boost::copy()`, über die eingegebenen Zahlen zu iterieren und sie auf **`std::cout`** auszugeben.
Sie können das Programm jederzeit mit Strg+C beenden.
Neben `boost::iterator_range` bietet Boost.Range auch eine Klasse `boost::sub_range` an. Diese Klasse ist von `boost::iterator_range` abgeleitet. So wie bei `boost::iterator_range` handelt es sich auch bei `boost::sub_range` um ein Template. `boost::sub_range` erwartet jedoch einen Rangetyp als Template-Parameter und keinen Iteratortyp. Das kann die Handhabung etwas vereinfachen. Sehen Sie sich dazu Beispiel 30.9 an.

Beispiel 30.9 Mit `boost::sub_range()` Ranges einfacher erstellen

```
#include <boost/range/algorithm.hpp>
#include <boost/range/iterator_range.hpp>
#include <boost/range/sub_range.hpp>
#include <array>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 2, 3, 4, 5}};
  boost::iterator_range<std::array<int, 6>::iterator> r1 =
    boost::random_shuffle(a);
  boost::sub_range<std::array<int, 6>> r2 =
    boost::random_shuffle(r1);
  boost::copy(r2, std::ostream_iterator<int>{std::cout, ","});
}
```

Einige von Boost.Range angebotene Algorithmen geben eine Range zurück – so zum Beispiel `boost::random_shuffle()`. Auch wenn dieser Algorithmus die Range, die als Parameter übergeben wird, direkt verändert, gibt er zusätzlich einen Verweis auf diese Range zurück.
Im Beispiel 30.9 wird `boost::random_shuffle()` zweimal aufgerufen und jedes Mal der Rückgabewert explizit gespeichert. Beim ersten Mal wird `boost::iterator_range` verwendet, beim zweiten Mal `boost::sub_range`. Beide Klassen unterscheiden sich lediglich im Template-Parameter. `boost::sub_range` hat nicht nur den Vorteil, dass die Klasse üblicherweise einfach zu instanziieren ist. Eine Range vom Typ `boost:`

:sub_range kennt auch einen Typ const_iterator, weil diese Klasse mit einer Range und nicht mit einem Iterator instanziiert wird.

Kapitel 31

Boost.Graph

Die Bibliothek Boost.Graph bietet Werkzeuge an, um mit Graphen zu arbeiten. Graphen sind zweidimensionale Punktwolken mit beliebig vielen Verbindungslinien zwischen Punkten. Ein U-Bahn-Plan ist ein Beispiel für einen Graph: U-Bahnhöfe sind Punkte, die über U-Bahnlinien verbunden sind.
Die Graphentheorie ist der Teilbereich der Mathematik, der sich mit Graphen beschäftigt. Dort wird zum Beispiel der Frage nachgegangen, wie man den kürzesten Weg zwischen zwei Punkten in einem Graphen findet. Dieses Problem muss jedes Navigationssystem lösen, wenn es einen Autofahrer auf dem kürzesten Weg zu seinem Ziel lotsen will. Graphen haben daher eine hohe praktische Relevanz. Viele Probleme lassen sich auf Graphen zurückführen und können mit ihnen gelöst werden.
Boost.Graph bietet Container an, um Graphen zu definieren. Viel wichtiger aber sind die von Boost.Graph zur Verfügung gestellten Algorithmen, um zum Beispiel kürzeste Wege zu finden. Wie Sie Container und Algorithmen von Boost.Graph verwenden, erfahren Sie im Folgenden.

31.1 Knoten und Kanten

Graphen bestehen aus Punkten und Verbindungslinien. Um einen Graph zu erstellen, muss daher angegeben werden, aus wie vielen Punkten er besteht und zwischen welchen Punkten Verbindungslinien verlaufen. Im Beispiel 31.1 soll ein erster einfacher Graph erstellt werden, der aus vier Punkten besteht, ohne dass diese verbunden werden.

Beispiel 31.1 Ein Graph vom Typ `boost::adjacency_list` mit vier Knoten

```
#include <boost/graph/adjacency_list.hpp>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v3 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v4 = boost::add_vertex(g);

  std::cout << v1 << ", " << v2 << ", " << v3 << ", " << v4 << '\n';
}
```

Boost.Graph bietet drei Container an, um Graphen zu definieren. Der wichtigste Container ist `boost::adjacency_list`, der in fast allen Beispielen in diesem Kapitel Verwendung findet. Um auf diese Klasse zugreifen zu können, müssen Sie die Headerdatei `boost/graph/adjacency_list.hpp` einbinden. Möchten Sie einen anderen Container verwenden, müssen Sie auf eine andere Headerdatei zugreifen. Es gibt keine Master-Headerdatei, um Zugriff auf alle Klassen und Funktionen von Boost.Graph zu erhalten.
`boost::adjacency_list` ist ein Template, das im Beispiel 31.1 mit Standardwerten instanziiert wird. Welche Template-Parameter Sie übergeben können, erfahren Sie später. Wie Sie sehen, ist die Klasse im Namensraum `boost` definiert. Alle für Anwender von Boost.Graph wichtigen Klassen und Funktionen sind in diesem Namensraum abgelegt.

Um dem Graphen vier Punkte hinzuzufügen, rufen Sie die Funktion `boost::add_vertex()` viermal auf. Diese Funktion erwartet als einzigen Parameter den Graphen, der den neuen Punkt erhalten soll.
Beachten Sie, dass es sich bei `boost::add_vertex()` um eine freistehende Funktion und nicht um eine Methode der Klasse `boost::adjacency_list` handelt. Sie finden in Boost.Graph sehr viele Funktionen, die man genauso gut als Methode hätte implementieren können. Boost.Graph will jedoch mehr eine generische als eine objektorientierte Bibliothek sein. Für Anwender von Boost.Graph macht es jedoch keinen Unterschied, ob eine Funktion oder Methode aufgerufen wird. Für einen besseren Überblick, welche freistehenden Funktionen mit `boost::adjacency_list` verwendet werden, führt die Dokumentation von `boost::adjacency_list` die entsprechenden Funktionen an.
Die Funktion, um einen Punkt einem Graphen hinzuzufügen, heißt `boost::add_vertex()`. In der Graphentheorie werden Punkte als Knoten oder Ecken bezeichnet. Ins Englische übersetzt wird daraus vertex – daher der Funktionsname.
`boost::add_vertex()` gibt ein Objekt vom Typ `boost::adjacency_list::vertex_descriptor` zurück. Dieses Objekt stellt den neu hinzugefügten Punkt im Graphen dar. Sie können wie im Beispiel 31.1 das Objekt direkt auf die Standardausgabe ausgeben. Wenn Sie Beispiel 31.1 ausführen, sehen Sie `0, 1, 2, 3`.
Im Beispiel 31.1 werden Punkte über positive Ganzzahlen identifiziert. Diese Zahlen sind Indizes in einen Vektor, der innerhalb von `boost::adjacency_list` verwendet wird. Es ist daher kein Wunder, dass `boost::add_vertex()` die Zahlen 0, 1, 2 und 3 zurückgibt – mit jedem Aufruf wird dem Vektor ein zusätzlicher Punkt hinzugefügt.
Es ist durchaus möglich, dass `boost::adjacency_list::vertex_descriptor` einen anderen Typ als std::size_t hat. Das hängt von den Template-Parametern ab, die an `boost::adjacency_list` übergeben werden. Standardmäßig verwendet `boost::adjacency_list` einen Vektor, um Punkte zu speichern.

Beispiel 31.2 Zugriff auf Knoten mit `boost::vertices()`

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;

  boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<boost::adjacency_list<>::vertex_iterator,
    boost::adjacency_list<>::vertex_iterator> vs = boost::vertices(g);

  std::copy(vs.first, vs.second,
    std::ostream_iterator<boost::adjacency_list<>::vertex_descriptor>{
      std::cout, "\n"});
}
```

Um auf alle Punkte in einem Graph zuzugreifen, kann `boost::vertices()` aufgerufen werden. Diese Funktion gibt zwei Iteratoren vom Typ `boost::adjacency_list::vertex_iterator` zurück, die auf den Anfang und das Ende der Punkte zeigen. Die Iteratoren werden in einem `std::pair` zurückgegeben. Im Beispiel 31.2 werden die Iteratoren verwendet, um die Punkte auf die Standardausgabe auszugeben. Dieses Beispiel gibt wie das vorherige die Zahlen 0, 1, 2 und 3 aus.
Beispiel 31.3 zeigt Ihnen, wie Punkte mit Linien verbunden werden.

Beispiel 31.3 Zugriff auf Kanten mit `boost::edges()`

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
```

```
{
  boost::adjacency_list<> g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<boost::adjacency_list<>::edge_descriptor, bool> p =
    boost::add_edge(v1, v2, g);
  std::cout.setf(std::ios::boolalpha);
  std::cout << p.second << '\n';

  p = boost::add_edge(v1, v2, g);
  std::cout << p.second << '\n';

  p = boost::add_edge(v2, v1, g);
  std::cout << p.second << '\n';

  std::pair<boost::adjacency_list<>::edge_iterator,
    boost::adjacency_list<>::edge_iterator> es = boost::edges(g);

  std::copy(es.first, es.second,
    std::ostream_iterator<boost::adjacency_list<>::edge_descriptor>{
      std::cout, "\n"});
}
```

Sie rufen die Funktion `boost::add_edge()` auf, um zwei Punkte in einem Graphen zu verbinden. Sie müssen dazu die Punkte und den Graphen als Parameter übergeben.
In der Graphentheorie werden Verbindungslinien als Kanten bezeichnet. Im Englischen wird daraus edge – daher der Funktionsname `boost::add_edge()`.
`boost::add_edge()` gibt ein `std::pair` zurück. Über **`first`** haben Sie Zugriff auf die Verbindungslinie. **`second`** ist eine bool-Variable, die angibt, ob die Verbindungslinie mit `boost::add_edge()` neu hinzugefügt wurde. Wenn Sie Beispiel 31.3 ausführen, sehen Sie, dass `p.second` jeweils auf `true` gesetzt ist und alle Aufrufe von `boost::add_edge()` dazu führen, dass eine neue Verbindungslinie dem Graphen hinzugefügt wird.
Über `boost::edges()` erhalten Sie Zugriff auf alle Verbindungslinien eines Graphen. So wie `boost::vertices()` gibt auch `boost::edges()` zwei Iteratoren zurück, die auf den Anfang und das Ende aller Verbindungslinien zeigen. Beispiel 31.3 gibt alle Verbindungslinien auf die Standardausgabe aus. Wenn Sie das Beispiel ausführen, erhalten Sie `(0,1)`, `(0,1)` und `(1,0)`.
Die Ausgabe zeigt, dass der Graph drei Verbindungslinien hat. Diese verlaufen alle zwischen den ersten beiden Punkten – die mit den Indizes 0 und 1. Die Ausgabe zeigt auch an, an welchem Punkt Verbindungslinien beginnen und enden. Zwei Verbindungslinien laufen vom ersten zum zweiten Punkt, eine Verbindungslinie in die umgekehrte Richtung. Die Richtung hängt davon ab, welcher Punkt jeweils als erster und zweiter Parameter an `boost::add_edge()` übergeben wird.
Es ist problemlos möglich, mehrere Verbindungslinien zwischen zwei Punkten aufzuspannen. Die Möglichkeit, die gleichen Verbindungslinien mehrfach zu definieren, kann jedoch ausgeschaltet werden.

Beispiel 31.4 `boost::adjacency_list` mit Selektoren

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
```

```
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<graph::edge_descriptor, bool> p =
    boost::add_edge(v1, v2, g);
  std::cout.setf(std::ios::boolalpha);
  std::cout << p.second << '\n';

  p = boost::add_edge(v1, v2, g);
  std::cout << p.second << '\n';

  p = boost::add_edge(v2, v1, g);
  std::cout << p.second << '\n';

  std::pair<graph::edge_iterator,
    graph::edge_iterator> es = boost::edges(g);

  std::copy(es.first, es.second,
    std::ostream_iterator<graph::edge_descriptor>{std::cout, "\n"});
}
```

Im Beispiel 31.4 wird `boost::adjacency_list` nicht mehr mit Standardwerten instanziiert. Stattdessen werden drei Template-Parameter übergeben, die *Selektoren* darstellen – deswegen enden ihre Namen mit einem S. Diese Selektoren bestimmen unter anderem, welche Typen innerhalb von `boost::adjacency_list` verwendet werden, um Punkte und Verbindungslinien zu speichern.
`boost::adjacency_list` verwendet standardmäßig `std::vector` für Punkte und Verbindungslinien. Indem im Beispiel 31.4 `boost::setS` als erster Template-Parameter angegeben wird, wird `std::set` als Container für Verbindungslinien ausgewählt. Weil `std::set` im Vergleich zu `std::vector` keine Duplikate zulässt, ist es im Beispiel 31.4 nicht möglich, mehrfach die gleiche Verbindungslinie mit `boost::add_edge()` hinzuzufügen. Wenn Sie das Beispielprogramm ausführen, sehen Sie, dass nur mehr `(0,1)` ausgegeben wird.
Über den zweiten Template-Parameter wird `boost::adjacency_list` mitgeteilt, welche Klasse für Punkte verwendet werden soll. Im Beispiel 31.4 ist dies `boost::vecS`. Über diesen Selektor wird `std::vector` ausgewählt. `boost::vecS` ist der Standardwert für den ersten und zweiten Template-Parameter von `boost::adjacency_list`.
Der dritte Template-Parameter gibt an, ob Verbindungslinien gerichtet oder ungerichtet sind. Standardmäßig verwendet `boost::adjacency_list` als Selektor `boost::directedS`. In diesem Fall sind Verbindungslinien gerichtet und können zum Beispiel als Pfeil dargestellt werden. Wird `boost::directedS` verwendet, können Verbindungslinien nur in eine Richtung laufen.
Im Beispiel 31.4 ist als Selektor `boost::undirectedS` angegeben, womit Verbindungslinien ungerichtet sind. Derartige Verbindungslinien können in beide Richtungen laufen. Der Start- und Endpunkt solcher Verbindungslinien spielt keine Rolle. Dies ist ein weiterer Grund, warum im Beispiel 31.4 lediglich eine Verbindungslinie erstellt wird. Beim dritten Aufruf von `boost::add_edge()` sind der Start- und Endpunkt ausgetauscht – die Verbindungslinie wird dennoch nicht dem Graphen hinzugefügt, da sie identisch mit der bereits beim ersten Aufruf von `boost::add_edge()` erstellten Linie ist.
Neben den erwähnten Selektoren bietet Boost.Graph weitere an. Zu diesen zählen zum Beispiel `boost::listS`, `boost::mapS` oder `boost::hash_setS`, um Container für Punkte und Verbindungslinien auszuwählen. Als weiterer Selektor für Verbindungslinien steht außerdem `boost::bidirectionalS` zur Verfügung, mit dem Verbindungslinien erstellt werden können, die doppelt gerichtet sind und in beide Richtungen laufen. Dieser Selektor ähnelt `boost::undirectedS`. Start- und Endpunkt sind bei `boost::bidirectionalS` jedoch entscheidend. Wenn Sie im Beispiel 31.4 `boost::bidirectionalS` verwenden, fügt der dritte Aufruf von `boost::add_edge()` dem Graphen eine neue Verbindungslinie hinzu.
Beispiel 31.5 zeigt Ihnen, wie Sie einfacher Punkte und Verbindungslinien erstellen.

Beispiel 31.5 Automatisch Knoten mit `boost::add_edge()` erstellen

```
#include <boost/graph/adjacency_list.hpp>
#include <tuple>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
```

```
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  enum { topLeft, topRight, bottomRight, bottomLeft };

  boost::add_edge(topLeft, topRight, g);
  boost::add_edge(topRight, bottomRight, g);
  boost::add_edge(bottomRight, bottomLeft, g);
  boost::add_edge(bottomLeft, topLeft, g);

  graph::edge_iterator it, end;
  std::tie(it, end) = boost::edges(g);
  std::copy(it, end,
    std::ostream_iterator<graph::edge_descriptor>{std::cout, "\n"});
}
```

Im Beispiel 31.5 wird ein Graph bestehend aus vier Punkten definiert. Um sich den Graphen besser als eine Karte mit vier Feldern vorstellen zu können, werden den Punkten Namen gegeben: `topLeft`, `topRight`, `bottomRight` und `bottomLeft`. Hinter diesen Namen stehen Zahlen, wie sie in vorherigen Beispielen von `boost::add_vertex()` zurückgegeben und als Indizes verwendet wurden.
Es ist möglich, den Graphen ohne einen Aufruf von `boost::add_vertex()` zu definieren. Boost.Graph fügt fehlende Punkte einem Graphen automatisch hinzu, wenn die an `boost::add_edge()` übergebenen Punkte nicht existieren. So führt der vierfache Aufruf von `boost::add_edge()` im Beispiel 31.5 dazu, dass nicht nur vier Verbindungslinien definiert werden, sondern der Graph gleichzeitig die für die Verbindungslinien notwendigen vier Punkte erhält.
Beachten Sie außerdem, wie mit `std::tie()` die Iteratoren, die `boost::edges()` in einem `std::pair` zurückgibt, direkt in **`it`** und **`end`** gespeichert werden. `std::tie()` ist seit C++11 Teil der Standardbibliothek.
Beim im Beispiel 31.5 erstellten Graphen handelt es sich um eine Karte, die aus vier Feldern besteht. Um von links oben nach rechts unten zu gelangen, kann entweder über das rechte obere oder das linke untere Feld gegangen werden. Es gibt keine Verbindung zu gegenüberliegenden Feldern, so dass nicht direkt von links oben nach rechts unten gelangt werden kann. Alle folgenden Beispiele in diesem Kapitel arbeiten mit diesem Graphen.

Beispiel 31.6 `boost::adjacent_vertices()` und `boost::out_edges()`

```
#include <boost/graph/adjacency_list.hpp>
#include <tuple>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  enum { topLeft, topRight, bottomRight, bottomLeft };

  boost::add_edge(topLeft, topRight, g);
  boost::add_edge(topRight, bottomRight, g);
  boost::add_edge(bottomRight, bottomLeft, g);
  boost::add_edge(bottomLeft, topLeft, g);

  graph::adjacency_iterator vit, vend;
  std::tie(vit, vend) = boost::adjacent_vertices(topLeft, g);
  std::copy(vit, vend,
    std::ostream_iterator<graph::vertex_descriptor>{std::cout, "\n"});

  graph::out_edge_iterator eit, eend;
  std::tie(eit, eend) = boost::out_edges(topLeft, g);
  std::for_each(eit, eend,
    [&g](graph::edge_descriptor it)
```

```
      { std::cout << boost::target(it, g) << '\n'; });
}
```

Im Beispiel 31.6 lernen Sie Funktionen kennen, um zusätzliche Informationen zu Punkten zu erhalten. So gibt `boost::adjacent_vertices()` ein Paar Iteratoren zurück, über die auf Punkte zugegriffen werden kann, mit denen der Punkt, der als Parameter an die Funktion übergeben ist, verbunden ist. `boost::out_edges()` können Sie aufrufen, wenn Sie auf die ausgehenden Verbindungslinien eines Punkts zugreifen möchten. Boost.Graph bietet auch eine Funktion `boost::in_edges()` an, um eingehende Verbindungslinien zu erhalten. Wird wie im Beispiel 31.6 mit ungerichteten Verbindungslinien gearbeitet, spielt es keine Rolle, welche dieser beiden Funktionen aufgerufen wird.
`boost::target()` gibt den Endpunkt einer Verbindungslinie zurück. Der Startpunkt wird mit `boost::source()` erhalten.
Beispiel 31.6 gibt zweimal 1 und 3 auf die Standardausgabe aus. Das sind die Indizes für das rechte obere und das linke untere Feld. Da das linke obere Feld an `boost::adjacent_vertices()` übergeben wird, werden die Indizes für die genannten Felder zurückgegeben und mit `std::copy()` auf die Standardausgabe ausgegeben. Das linke obere Feld wird auch an `boost::out_edges()` übergeben, so dass auf die ausgehenden Verbindungslinien zugegriffen werden kann. Da in `std::for_each()` auf `boost::target()` zugegriffen wird, werden wieder die Indizes des rechten oberen und linken unteren Feldes ausgegeben.
Beispiel 31.7 zeigt Ihnen, wie Sie einen Graphen beim Instanziieren von `boost::adjacency_list` definieren können, ohne für jede Verbindungslinie `boost::add_edge()` aufrufen zu müssen.

Beispiel 31.7 Verbindungslinien beim Instanziieren definieren

```
#include <boost/graph/adjacency_list.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  std::cout << boost::num_vertices(g) << '\n';
  std::cout << boost::num_edges(g) << '\n';

  g.clear();
}
```

Sie können an den Konstruktor von `boost::adjacency_list` Iteratoren übergeben, die auf Objekte vom Typ std::pair<int, int> zeigen, die Verbindungslinien definieren. Übergeben Sie Iteratoren, müssen Sie außerdem als dritten Parameter angeben, wie viele Punkte der Graph erhalten soll. Es werden auf alle Fälle die Punkte definiert, die für die Definition der Verbindungslinien notwendig sind. Über den dritten Parameter können Sie dem Graphen zusätzliche Punkte hinzufügen, die nicht mit anderen Punkten verbunden sind.
Sie sehen im Beispiel 31.7 weitere Funktionen wie `boost::num_vertices()` und `boost::num_edges()`, mit denen die Anzahl der Punkte und Verbindungslinien erhalten werden kann. Wenn Sie das Beispielprogramm ausführen, wird zweimal 4 auf die Standardausgabe ausgegeben.
Im Beispiel 31.7 wird außerdem `boost::adjacency_list::clear()` aufgerufen. Mit dieser Methode können alle Punkte und Verbindungslinien entfernt werden. Hierbei handelt es sich tatsächlich um eine Methode der Klasse `boost::adjacency_list` und nicht um eine freistehende Funktion.

31.2 Algorithmen

Nachdem Sie gesehen haben, wie Sie eigene Graphen definieren, lernen Sie im Folgenden Algorithmen kennen. Algorithmen von Boost.Graph ähneln denen aus der Standardbibliothek – sie sind generisch und höchst flexibel. Es erschließt sich aber nicht immer auf den ersten Blick, wie sie verwendet werden.

Beispiel 31.8 Punkte von innen nach außen mit `breadth_first_search()` besuchen

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iterator>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> distances{{0}};

  boost::breadth_first_search(g, topLeft,
    boost::visitor(
      boost::make_bfs_visitor(
        boost::record_distances(distances.begin(),
          boost::on_tree_edge{}))));

  std::copy(distances.begin(), distances.end(),
    std::ostream_iterator<int>{std::cout, "\n"});
}
```

Im Beispiel 31.8 wird der Algorithmus `boost::breadth_first_search()` verwendet, um Punkte von innen nach außen zu besuchen. Der Algorithmus beginnt an dem Punkt, der als zweiter Parameter übergeben wird, und besucht zuerst alle Punkte, die von diesem Punkt direkt erreicht werden können. `boost::breadth_first_search()` funktioniert wie eine sich in alle Richtungen ausbreitende Wasserwelle.

`boost::breadth_first_search()` gibt jedoch kein bestimmtes Ergebnis zurück. Der Algorithmus besucht Punkte – sonst nichts. Ob bei diesen Besuchen Daten ermittelt und gespeichert werden sollen, hängt davon ab, welche *Besucher* Sie übergeben.

Besucher sind Objekte, deren Methoden aufgerufen werden, wenn ein Punkt besucht wird. Indem Sie Besucher an einen Algorithmus wie `boost::breadth_first_search()` übergeben, legen Sie fest, was passieren soll, wenn ein Punkt besucht wird. Besucher entsprechen Funktionsobjekten, die an Algorithmen aus der Standardbibliothek übergeben werden können.

Im Beispiel 31.8 wird ein Besucher verwendet, der Distanzen aufzeichnen kann. Als Distanz wird die Anzahl der Verbindungslinien bezeichnet, die durchlaufen werden müssen, um zu einem bestimmten Punkt zu gelangen – ausgehend von dem Punkt, der als zweiter Parameter an `boost::breadth_first_search()` übergeben wird.

Boost.Graph bietet die Hilfsfunktion `boost::record_distances()` an, um das entsprechende Objekt zu erstellen. An diese Funktion muss eine *Property-Map* und ein *Tag* übergeben werden.

Property-Maps sind Objekte, die Eigenschaften zu Punkten oder Verbindungslinien in einem Graphen speichern

können. Das Konzept der Property-Maps ist in der Bibliothek Boost.Graph beschrieben. Ein Zeiger oder ein Iterator auf einen entsprechenden Container wird automatisch als Anfang einer Property-Map interpretiert, so dass Sie sich nicht im Detail mit Property-Maps beschäftigen müssen. Im Beispiel 31.8 wird mit `distances.begin()` der Anfang des Arrays **`distances`** an `boost::record_distances()` übergeben, womit das Array als Property-Map für Punkte verwendet wird. Dabei ist wichtig, dass die Länge des Arrays der Anzahl der Punkte im Graphen entspricht. Schließlich soll für jeden Punkt im Graphen die Distanz zum Ausgangspunkt gespeichert werden.
Beachten Sie, dass **`distances`** auf dem Typ `boost::array` und nicht auf `std::array` basiert. `std::array` würde zu einem Compilerfehler führen.
Abhängig vom verwendeten Algorithmus gibt es verschiedene Ereignisse, wenn ein Punkt besucht wird. Über den zweiten Parameter, der an `boost::record_distances()` übergeben wird, wird angegeben, zu welchem Ereignis der Besucher informiert werden soll. Boost.Graph definiert hierfür Tags, bei denen es sich um leere Klassen handelt, deren einziger Sinn darin besteht, ein Ereignis auszuwählen. So ist im Beispiel 31.8 mit dem Tag `boost::on_tree_edge` angegeben, dass die Distanz dann aufgezeichnet werden soll, wenn über eine Verbindungslinie ein neuer Punkt gefunden wurde.
Ereignisse sind abhängig vom verwendeten Algorithmus. Welche Ereignisse ein Algorithmus unterstützt und welche Tags Sie verwenden können, müssen Sie jeweils in der Dokumentation des Algorithmus nachschlagen.
Ein Besucher wie der, der von `boost::record_distances()` erstellt wird, ist unabhängig vom Algorithmus. Sie können `boost::record_distances()` also durchaus im Zusammenhang mit anderen Algorithmen verwenden. Die Verbindung zwischen dem jeweils verwendeten Algorithmus und dem Besucher findet über einen Adapter statt. Im Beispiel 31.8 wird dieser Adapter über `boost::make_bfs_visitor()` erstellt. Diese Hilfsfunktion gibt ein Besucherobjekt zurück, wie es vom Algorithmus `boost::breadth_first_search()` erwartet wird. Dieses Besucherobjekt bietet Methoden an, die genau zu den Ereignissen passen, die der jeweilige Algorithmus unterstützt. So definiert das Besucherobjekt, das von `boost::make_bfs_visitor()` zurückgegeben wird, zum Beispiel eine Methode namens `tree_edge()`. Wird an `boost::make_bfs_visitor()` ein Besucher übergeben, der wie im Beispiel 31.8 mit dem Tag `boost::on_tree_edge` definiert wird, wird dieser Besucher dann aufgerufen, wenn `tree_edge()` aufgerufen wird. Auf diese Weise ist es möglich, Besucher mit unterschiedlichen Algorithmen zu verwenden, ohne dass Besucher wie der, der mit `boost::record_distances()` erstellt wird, sämtliche Methoden definieren müssen, wie sie von allen nur erdenklichen Algorithmen benötigt werden.
Der Besucher-Adapter, der von `boost::make_bfs_visitor()` zurückgegeben wird, kann nicht direkt als dritter Parameter an `boost::breadth_first_search()` übergeben werden. Er muss mit `boost::visitor()` verpackt werden, um als dritter Parameter akzeptiert werden zu können.
Algorithmen wie `boost::breadth_first_search()` gibt es in zwei Varianten: Eine Variante erwartet, dass sämtliche vom Algorithmus erwarteten Parameter einzeln übergeben werden. Die andere Variante unterstützt etwas Ähnliches wie Parameter mit Namen. Diese zweite Variante ist üblicherweise einfacher zu verwenden, weil nur die Parameter übergeben werden müssen, an denen Interesse besteht. Viele Parameter müssen oft nicht übergeben werden, weil Algorithmen Standardwerte verwenden.
Die im Beispiel 31.8 verwendete Variante von `boost::breadth_first_search()` erwartet, dass Parameter mit Namen übergeben werden – abgesehen von den ersten beiden Parametern, die den Graphen und den Ausgangspunkt darstellen. Der dritte Parameter von `boost::breadth_first_search()` kann aber quasi alles Mögliche sein. Im Beispiel 31.8 soll ein Besucher übergeben werden. Damit das funktioniert, wird der Besucher-Adapter, der von `boost::make_bfs_visitor()` zurückgegeben wird, mit `boost::visitor()` benannt. Es ist nun klar, dass es sich beim dritten Parameter um einen Besucher handelt. Sie werden in nachfolgenden Beispielen sehen, wie Sie andere oder mehrere Parameter mit Namen an dritter Stelle an `boost::breadth_first_search()` übergeben.
Wenn Sie Beispiel 31.8 ausführen, werden Ihnen die Zahlen 0, 1, 2 und 1 ausgegeben. Die Zahlen stellen die Distanzen zu anderen Punkten dar – ausgehend von links oben. Zum Feld rechts oben – das mit dem Index 1 – ist lediglich ein Schritt nötig. Zum Feld rechts unten – das mit dem Index 2 – sind zwei Schritte nötig. Zum Feld links unten – das mit dem Index 3 – ist wiederum nur ein Schritt nötig. Die Zahl 0, die zuerst ausgegeben wird, bezieht sich auf das Feld links oben. Da es sich dabei um den Ausgangspunkt handelt, der an `boost::breadth_first_search()` übergeben wurde, ist die Schrittzahl entsprechend 0.
Beachten Sie, dass Sie alle Felder im Array **`distances`** initialisieren müssen. `boost::breadth_first_search()` setzt die Felder im Array nicht direkt, sondern erhöht lediglich die im Array vorgefundenen Werte.
Nachdem Sie im Beispiel 31.8 gesehen haben, wie Sie die Distanz zu Punkten berechnen können, erfahren Sie im Folgenden, wie Sie den kürzesten Weg zu Punkten finden.

Beispiel 31.9 Wegstrecken mit `breadth_first_search()` finden

```
#include <boost/graph/adjacency_list.hpp>
```

```
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> predecessors;
  predecessors[bottomRight] = bottomRight;

  boost::breadth_first_search(g, bottomRight,
    boost::visitor(
      boost::make_bfs_visitor(
        boost::record_predecessors(predecessors.begin(),
          boost::on_tree_edge{}))));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
  std::cout << p << '\n';
}
```

Beispiel 31.9 gibt 0, 1 und 2 auf die Standardausgabe aus. Dies ist der kürzeste Weg von links oben nach rechts unten. Er führt über das Feld rechts oben, wobei der Weg über links unten natürlich genauso kurz wäre.
Der kürzeste Weg wird mit dem bereits bekannten Algorithmus `boost::breadth_first_search()` berechnet. Wie ebenfalls bereits bekannt macht dieser Algorithmus nichts anderes als Punkte zu besuchen. Um Wegbeschreibungen zu erhalten, muss deswegen ein geeigneter Besucher verwendet werden. Dieser wird im Beispiel 31.9 mit `boost::record_predecessors()` erhalten.
`boost::record_predecessors()` gibt einen Besucher zurück, der jeweils den Vorgänger eines Punkts speichert. Immer dann, wenn `boost::breadth_first_search()` einen neuen Punkt besucht, wird in der Property-Map, die an `boost::record_predecessors()` übergeben wird, der Punkt gespeichert, von dem aus der neue Punkt gefunden wurde. Da `boost::breadth_first_search()` von innen nach außen vorgeht, wird auf diese Weise die kürzeste Wegbeschreibung zu allen Punkten erhalten – ausgehend von dem Punkt, der als zweiter Parameter an `boost::breadth_first_search()` übergeben wird. Beispiel 31.9 ermittelt die kürzesten Wege von allen Punkten im Graphen nach rechts unten.
Nachdem `boost::breadth_first_search()` aufgerufen wurde, enthält die Property-Map **`predecessors`** die jeweiligen Vorgänger jedes Punkts. Um das erste Feld zu finden, zu dem man sich von links oben nach rechts unten bewegen muss, wird mit dem Index 0 – das ist der Index des linken oberen Felds – auf **`predecessors`** zugegriffen. Der Wert an dieser Stelle in der Property-Map ist 1. Dies bedeutet, dass das nächste Feld das rechte obere ist. Der Zugriff mit dem Index 1 auf **`predecessors`** ergibt das nächste Feld. Das ist in diesem Beispiel bereits das rechte untere – das mit dem Index 2. Auf diese Weise können in beliebig großen Graphen iterativ die Punkte gefunden werden, anhand derer man sich von einem Start- zu einem Endpunkt fortbewegen muss.

Beispiel 31.10 zeigt, wie `boost::breadth_first_search()` mit zwei Besuchern verwendet wird.

Beispiel 31.10 Distanzen und Wegstrecken mit `breadth_first_search()` finden

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> distances{{0}};
  boost::array<int, 4> predecessors;
  predecessors[bottomRight] = bottomRight;

  boost::breadth_first_search(g, bottomRight,
    boost::visitor(
      boost::make_bfs_visitor(
        std::make_pair(
          boost::record_distances(distances.begin(),
            boost::on_tree_edge()),
          boost::record_predecessors(predecessors.begin(),
            boost::on_tree_edge{})))));

  std::for_each(distances.begin(), distances.end(),
    [](int d){ std::cout << d << '\n'; });

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
  std::cout << p << '\n';
}
```

Um zwei Besucher zu verwenden, müssen sie wie im Beispiel 31.10 mit `std::make_pair()` zu einem Paar verknüpft werden. Sollen mehr als zwei Besucher verwendet werden, müssen Sie die Paare verschachteln. Beispiel 31.10 macht das Gleiche wie Beispiel 31.8 und Beispiel 31.9 zusammen.

Der Algorithmus `boost::breadth_first_search()` kann nur dann verwendet werden, wenn es keine unterschiedliche Gewichtung zwischen Verbindungen gibt. Verbindungen zwischen zwei Punkten können also zum Beispiel gleich schnell durchlaufen werden. Werden Gewichtungen vorgenommen, muss ein anderer Algorithmus verwendet werden, um kürzeste Wege zu finden.

Beispiel 31.11 Wegstrecken mit `dijkstra_shortest_paths()` finden

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
```

```
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    boost::property<boost::edge_weight_t, int>> graph;

  std::array<int, 4> weights{{2, 1, 1, 1}};

  graph g{edges.begin(), edges.end(), weights.begin(), 4};

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Im Beispiel 31.11 wird `boost::dijkstra_shortest_paths()` verwendet, um die kürzesten Wegstrecken nach rechts unten zu finden. Dieser Algorithmus wird eingesetzt, wenn Verbindungen unterschiedlich gewichtet sind. So wird für Beispiel 31.11 angenommen, dass es doppelt so lange dauert, von links oben nach rechts oben zu gelangen, als jede andere Verbindung zu überqueren.
Bevor `boost::dijkstra_shortest_paths()` verwendet werden kann, muss den Verbindungen ein entsprechendes Gewicht zugewiesen werden. Dies geschieht mit Hilfe des Arrays **`weights`**. Die Elemente im Array entsprechen den Verbindungslinien im Graphen. Da die erste Verbindungslinie von links oben nach rechts oben geht, ist das erste Element in **`weights`** auf einen doppelt so großen Wert wie alle anderen Elemente gesetzt.
Um die Gewichte den Verbindungslinien zuzuweisen, wird der Iterator auf den Anfang des Arrays **`weights`** als dritter Parameter an den Konstruktor des Graphen übergeben. Dieser dritte Parameter kann verwendet werden, um Verbindungseigenschaften zu initialisieren. Dies wiederum funktioniert nur, wenn Eigenschaften für Verbindungen definiert wurden.
Wenn Sie sich Beispiel 31.11 näher ansehen, stellen Sie fest, dass zusätzliche Template-Parameter an `boost::adjacency_list` übergeben werden. Der vierte und fünfte Template-Parameter geben an, ob und welche Eigenschaften Punkte und Verbindungslinien haben. Es ist also nicht nur möglich, Verbindungslinien Eigenschaften zuzuweisen, sondern auch Punkten.
Standardmäßig verwendet `boost::adjacency_list` `boost::no_property`, um anzugeben, dass weder Punkte noch Verbindungslinien Eigenschaften haben. Für den vierten Parameter im Beispiel 31.11 ist `boost::no_property` angegeben, um keine Eigenschaften für Punkte zu definieren. Der fünfte Parameter verwendet jedoch `boost::property`, um eine *gebündelte Eigenschaft* zu erstellen.
Gebündelte Eigenschaften sind Eigenschaften, die intern im Graphen gespeichert werden. Da es möglich ist, beliebig viele gebündelte Eigenschaften zu definieren, erwartet `boost::property` einen Tag, um die Eigenschaft definieren zu können. Boost.Graph stellt einige Tags wie `boost::edge_weight_t` bereit, um oft benö-

tigte Eigenschaften zu definieren, die automatisch von Algorithmen erkannt und verwendet werden. Als zweiten Template-Parameter wird an `boost::property` der Typ übergeben, der für die Eigenschaft verwendet werden soll. Im Beispiel 31.11 sollen Gewichte über int-Zahlen definiert werden.
Beispiel 31.11 funktioniert, weil `boost::dijkstra_shortest_paths()` automatisch auf die mit Verbindungslinien gebündelte Eigenschaft vom Typ `boost::edge_weight_t` zugreift.
Beachten Sie, dass `boost::dijkstra_shortest_paths()` als dritten Parameter keinen Besucher bekommt. Dieser Algorithmus besucht nicht nur einfach Punkte, sondern hat das Ziel, den jeweils kürzesten Weg zu ermitteln – deswegen heißt der Algorithmus `boost::dijkstra_shortest_paths()`. Sie müssen sich also nicht überlegen, mit welchem Besucher Sie wie auf welches Ereignis reagieren wollen. Sie müssen lediglich einen Container übergeben, der von `boost::dijkstra_shortest_paths()` verwendet wird, um den Vorgänger zu jedem Punkt abzulegen. Wenn Sie wie im Beispiel 31.11 die Variante von `boost::dijkstra_shortest_paths()` verwenden, die Parameter mit Namen erwartet, übergeben Sie den Container mit `boost::predecessor_map()`. Es handelt sich hierbei um eine Hilfsfunktion, die wie andere einen Zeiger oder Iterator auf den Anfang eines Arrays erwartet.
Wenn Sie Beispiel 31.11 ausführen, wird 0, 3 und 2 ausgegeben: Der kürzeste Weg von links oben nach rechts unten führt über das linke untere Feld. Der Weg über das rechte obere Feld würde über eine Verbindung führen, die eine doppelte Gewichtung erhalten hat als alle anderen.

Beispiel 31.12 Benutzerdefinierte Eigenschaften mit `dijkstra_shortest_paths()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    edge_properties> graph;

  graph g{edges.begin(), edges.end(), 4};

  graph::edge_iterator it, end;
  boost::tie(it, end) = boost::edges(g);
  g[*it].weight = 2;
  g[*++it].weight = 1;
  g[*++it].weight = 1;
  g[*++it].weight = 1;

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()).
    weight_map(boost::get(&edge_properties::weight, g)));

  int p = topLeft;
  while (p != bottomRight)
```

```
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Beispiel 31.12 funktioniert wie das vorherige und gibt das gleiche Ergebnis aus. Der Unterschied ist, dass nun nicht mehr eine vordefinierte Eigenschaft, sondern eine benutzerdefinierte Eigenschaft verwendet wird. So wird im Beispiel 31.12 nicht mehr auf `boost::property` zugegriffen. Mit `edge_properties` wird eine eigene Klasse als Eigenschaft für Verbindungen angegeben.
`edge_properties` definiert eine Variable **`weight`**, um das Gewicht einer Verbindung zu speichern. Es könnten beliebig viele weitere Variablen definiert werden, wenn zusätzliche Eigenschaften für Verbindungen notwendig sind.
Sie können auf benutzerdefinierte Eigenschaften zugreifen, wenn Sie den Deskriptor für Verbindungen als Index für den Graphen verwenden. Der Graph wird also wie ein Array behandelt. Die Deskriptoren wiederum erhalten Sie über die Iteratoren für Verbindungen, die von `boost::edges()` zurückgegeben werden. Auf diese Weise ist es möglich, jeder Verbindung ein Gewicht zuzuweisen.
Damit `boost::dijkstra_shortest_paths()` versteht, dass sich in der Variablen **`weight`** in `edge_properties` die Gewichte für Verbindungen befinden, muss jedoch ein zusätzlicher Parameter mit Namen übergeben werden. Dies geschieht mit Hilfe von `weight_map()`. Beachten Sie, dass es sich hierbei um eine Methode des Objekts handelt, das von `boost::predecessor_map()` zurückgegeben wird. Es existiert auch eine freistehende Funktion `boost::weight_map()`. Sollen mehrere Parameter mit Namen übergeben werden, müssen jedoch mehrfach Methoden des Objekts aufgerufen werden, das beim Aufruf der freistehenden Funktion zurückgegeben wird. Auf diese Weise werden alle Parameter in das eine Objekt gepackt, das dann dem Algorithmus übergeben wird.
Um `boost::dijkstra_shortest_paths()` mitzuteilen, dass **`weight`** in `edge_properties` die Gewichte enthält, wird der Zeiger auf diese Eigenschaft übergeben. Er wird nicht direkt an `weight_map()` übergeben, sondern in einem Objekt, das mit `boost::get()` erstellt werden muss. An dieser Stelle ist der Code komplett und `boost::dijkstra_shortest_paths()` versteht, wie auf welche Eigenschaft zugegriffen werden muss, um Gewichte zu erhalten.

Beispiel 31.13 Benutzerdefinierte Eigenschaften beim Instanziieren initialisieren

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    edge_properties> graph;

  boost::array<edge_properties, 4> props{{2, 1, 1, 1}};
```

```
  graph g{edges.begin(), edges.end(), props.begin(), 4};

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()).
    weight_map(boost::get(&edge_properties::weight, g)));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Wenn Sie mit benutzerdefinierten Eigenschaften arbeiten, können Sie diese auch beim Instanziieren des Graphen initialisieren. Sie müssen lediglich an den Konstruktor von `boost::adjacency_list` als dritten Parameter einen Iterator übergeben, der auf Objekte vom Typ der benutzerdefinierten Eigenschaft zeigt. Es ist also nicht notwendig, über Deskriptoren einzeln auf Eigenschaften von Verbindungen zuzugreifen. Beispiel 31.13 funktioniert genauso wie das vorherige Beispiel und gibt die gleichen Ergebnisse aus.

Beispiel 31.14 Zufällige Wegstrecken mit `random_spanning_tree()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/random_spanning_tree.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <random>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS> graph;

  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> predecessors;

  std::mt19937 gen{static_cast<uint32_t>(std::time(0))};
  boost::random_spanning_tree(g, gen,
    boost::predecessor_map(predecessors.begin()).
    root_vertex(bottomLeft));

  int p = topRight;
```

```
  while (p != -1)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
}
```

Abschließend wird Ihnen im Beispiel 31.14 ein Algorithmus vorgestellt, der zufällige Wegstrecken zurückgibt. `boost::random_spanning_tree()` funktioniert ähnlich wie `boost::dijkstra_shortest_paths()` und gibt die Vorgänger von Punkten in einem Container zurück, der mit `boost::predecessor_map` übergeben werden muss. Im Gegensatz zu `boost::dijkstra_shortest_paths()` wird der Punkt, zu dem die zufälligen Wegstrecken ermittelt werden sollen, nicht direkt an `boost::random_spanning_tree()` übergeben. Sie müssen ihn stattdessen als Parameter mit Namen übergeben. Deswegen wird für das Objekt vom Typ `boost::predecessor_map` `root_vertex()` aufgerufen. Im Beispiel 31.14 sollen also zufällige Wege zum Punkt links unten ermittelt werden.
Da `boost::random_spanning_tree()` einen zufälligen Weg finden soll, erwartet der Algorithmus als zweiten Parameter einen Zufallsgenerator. Im Beispiel 31.14 wird ein Objekt vom Typ `std::mt19937` übergeben. Diese Klasse ist seit C++11 Teil der Standardbibliothek. Sie können auch einen Zufallsgenerator von Boost.Random verwenden.
Wenn Sie Beispiel 31.14 ausführen, wird Ihnen entweder 1, 0 und 3 oder 1, 2 und 3 ausgegeben. 1 ist das rechte obere Feld, 3 das linke untere Feld. Da man sowohl über das linke obere als auch das rechte untere Feld nach links unten gelangen kann, hängt es vom Zufall ab, welcher Weg von `boost::random_spanning_tree()` zurückgegeben wird.

31.3 Graphencontainer

Alle Beispiele in diesem Kapitel verwendeten zur Definition von Graphen die Klasse `boost::adjacency_list`. Im Folgenden werden die anderen beiden von Boost.Graph angebotenen Klassen `boost::adjacency_matrix` und `boost::compressed_sparse_row_graph` vorgestellt.

> Anmerkung
>
> In Boost 1.57.0 fehlt eine Headerdatei in `boost/graph/adjacency_matrix.hpp`. Um Beispiel 31.15 mit Boost 1.57.0 erfolgreich zu kompilieren, müssen Sie die Headerdatei `boost/functional/hash.hpp` vor `boost/graph/adjacency_matrix.hpp` einbinden.

Beispiel 31.15 Graphen mit `boost::adjacency_matrix`

```
#include <boost/graph/adjacency_matrix.hpp>
#include <array>
#include <utility>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_matrix<boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};
}
```

Wie im Beispiel 31.15 zu sehen, wird `boost::adjacency_matrix` grundsätzlich genauso verwendet wie `boost::adjacency_list`. Ein Unterschied ist jedoch, dass die ersten beiden Template-Parameter zur Auswahl von Containern für Punkte und Verbindungslinien bei `boost::adjacency_matrix` nicht existieren. Im Zusammenhang mit `boost::adjacency_matrix` werden also keine Selektoren wie `boost::vecS` oder `boost::setS` verwendet. `boost::adjacency_matrix` speichert den Graphen als Matrix, so dass die interne Struktur vorgegeben ist. Wenn Sie sich die Matrix als zweidimensionale Tabelle vorstellen: Die Tabelle hat genauso viele Reihen und Spalten wie der Graph Punkte hat. Verbindungslinien werden erstellt, indem in der entsprechenden Zelle, die der Schnittpunkt zweier Punkte ist, eine Markierung vorgenommen wird.
Die Struktur von `boost::adjacency_matrix` erlaubt es, Verbindungen sehr schnell zu erstellen und zu entfernen. Dafür ist der Speicherbedarf höher. Als Grundregel gilt: Verwenden Sie `boost::adjacency_list`, wenn es relativ wenige Verbindungen im Vergleich zu Punkten gibt. Je mehr Verbindungen, umso mehr ergibt es Sinn, `boost::adjacency_matrix` zu verwenden.

Beispiel 31.16 Graphen mit `boost::compressed_sparse_row_graph`

```
#include <boost/graph/compressed_sparse_row_graph.hpp>
#include <array>
#include <utility>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::compressed_sparse_row_graph<boost::bidirectionalS> graph;
  graph g{boost::edges_are_unsorted_multi_pass, edges.begin(),
    edges.end(), 4};
}
```

Die Klasse `boost::compressed_sparse_row_graph` wird wie im Beispiel 31.16 zu sehen ähnlich wie `boost::adjacency_list` und `boost::adjacency_matrix` verwendet. Der entscheidende Unterschied ist, dass keine Änderungen bei Graphen vom Typ `boost::compressed_sparse_row_graph` möglich sind. Punkte und Verbindungen können nach der Instanziierung des Graphen weder hinzugefügt noch entfernt werden. `boost::compressed_sparse_row_graph` ergibt daher nur Sinn, wenn mit Graphen gearbeitet wird, deren Definition unveränderlich ist.
Beachten Sie außerdem, dass `boost::compressed_sparse_row_graph` ausschließlich gerichtete Verbindungslinien unterstützt. Sie können `boost::compressed_sparse_row_graph` nicht mit dem Template-Parameter `boost::undirectedS` instanziieren.
Der Vorteil von `boost::compressed_sparse_row_graph` ist, dass Graphen kompakter gespeichert werden können. `boost::compressed_sparse_row_graph` bietet sich daher bei sehr großen Graphen an, um den Speicherbedarf gering zu halten.

Teil VI

Kommunikation

Boost.Asio und Boost.Interprocess ermöglichen es Programmen, miteinander zu kommunizieren.

- Boost.Asio ist die Bibliothek, die Sie verwenden, wenn Sie über ein Netzwerk kommunizieren möchten. Die Bibliothek kann jedoch nicht nur zur Netzwerkkommunikation verwendet werden. Asio steht für asynchrones Input/Output. Sie können diese Bibliothek einsetzen, um Daten asynchron zu verarbeiten. Dies ist üblicherweise dann möglich, wenn Ihr Programm mit Geräten kommuniziert, die Aufgaben parallel zu Code in Ihrem Programm ausführen können – wie beispielsweise Netzwerkkarten.
- Boost.Interprocess verwenden Sie, wenn über Shared Memory kommuniziert werden soll.

Kapitel 32

Boost.Asio

In diesem Kapitel lernen Sie die Bibliothek Boost.Asio kennen. Asio steht für asynchrones Input/Output. Mit dieser Bibliothek ist es möglich, Daten asynchron zu verarbeiten. Asynchron bedeutet, dass Operationen angestoßen werden, ohne auf ihren Abschluss zu warten. Stattdessen informiert Boost.Asio ein Programm, wenn eine Operation abgeschlossen wurde. Der Vorteil ist, dass in der Zwischenzeit andere Operationen ausgeführt werden können.
Mit Boost.Thread existiert eine andere Boost-Bibliothek, die es ermöglicht, Operationen gleichzeitig auszuführen. Der entscheidende Unterschied zwischen Boost.Thread und Boost.Asio ist, dass Sie mit Boost.Thread auf Ressourcen innerhalb eines Programms und mit Boost.Asio auf Ressourcen außerhalb eines Programms zugreifen. Haben Sie beispielsweise eine Funktion entwickelt, die eine zeitaufwändige Berechnung durchführen soll, können Sie diese in einen Thread packen und sie auf einem anderen Prozessorkern ausführen. Über Threads greifen Sie auf Prozessorkerne zu, die Ihren Code ausführen. Prozessorkerne stellen aus Sicht Ihres Programms interne Ressourcen dar. Möchten Sie auf Ressourcen zugreifen, die sich außerhalb Ihres Programms befinden, verwenden Sie Boost.Asio.
Ein Beispiel für externe Ressourcen sind Netzwerkverbindungen. Wenn Daten gesendet oder empfangen werden sollen, wird der Netzwerkkarte mitgeteilt, dass sie entsprechende Operationen ausführen soll. Zum Versand wird ihr ein Zeiger auf einen Speicherbereich übergeben, in dem die zu sendenden Daten liegen. Zum Empfang wird ihr ebenfalls ein Zeiger auf einen Speicherbereich übergeben, in den die zu empfangenen Daten abgelegt werden sollen. Da die Netzwerkkarte aus Sicht Ihres Programms eine externe Ressource ist, kann sie die entsprechenden Operationen unabhängig ausführen. Die Netzwerkkarte benötigt zur Ausführung der Operationen Zeit – Zeit, die Sie in Ihrem Programm nutzen können, um andere Operationen auszuführen. Boost.Asio ermöglicht es Ihnen, Geräte effizienter zu nutzen, indem Sie von ihrer Eigenschaft profitieren, parallel zum Prozessor Operationen ausführen zu können.
Der Versand und Empfang von Daten über Netzwerkverbindungen ist in Boost.Asio als asynchrone Operation implementiert. Eine asynchrone Operation ist vergleichbar mit einer Funktion, die nach einem Aufruf zurückkehrt, ohne ein Ergebnis zurückgeben zu können. Das Ergebnis wird später nachgereicht.
Eine asynchrone Operation wird in einem ersten Schritt initiiert. Ist die asynchrone Operation beendet, wird das Programm in einem zweiten Schritt über den Abschluss informiert. Die Aufteilung in Initiierung und Benachrichtigung über ihren Abschluss macht es möglich, auf externe Ressourcen ohne blockierende Funktionen zuzugreifen.

32.1 I/O Services und I/O Objekte

Programme, die Boost.Asio zur asynchronen Datenverarbeitung verwenden, basieren auf *I/O Services* und *I/O Objekten*. I/O Services abstrahieren Betriebssystemschnittstellen, die Daten asynchron verarbeiten können. I/O Objekte stellen Funktionen zur Verfügung, um asynchrone Operationen zu initiieren. Diese beiden Konzepte sind notwendig, um Aufgaben klar zu verteilen: I/O Services sind betriebssystemorientiert, I/O Objekte aufgabenorientiert.
Als Anwender von Boost.Asio kommen Sie mit I/O Services üblicherweise nicht direkt in Kontakt. I/O Services werden von einem *I/O Serviceobjekt* verwaltet. Sie können sich das I/O Serviceobjekt als Registrierungsdatenbank vorstellen, in der I/O Services automatisch registriert werden, wenn sie gebraucht werden. Jedes I/O Objekt kennt seinen I/O Service und erhält Zugriff auf diesen über das I/O Serviceobjekt. Sie stellen I/O Objekten lediglich ein I/O Serviceobjekt zur Verfügung – wann wie welche I/O Services registriert werden, geschieht automa-

tisch.
Boost.Asio definiert mit `boost::asio::io_service` eine einzige Klasse für das I/O Serviceobjekt. Jedes Programm, das auf Boost.Asio basiert, verwendet ein Objekt vom Typ `boost::asio::io_service`. Das kann der Einfachheit halber global erstellt werden.
Während es lediglich eine Klasse für das I/O Serviceobjekt gibt, gibt es zahlreiche Klassen für I/O Objekte. Da I/O Objekte aufgabenorientiert sind, hängt es von den Aufgaben ab, die erledigt werden sollen, welche Klassen instanziiert werden müssen. Sollen zum Beispiel Daten über eine TCP/IP-Verbindung gesendet oder empfangen werden, kann ein I/O Objekt `boost::asio::ip::tcp::socket` verwendet werden. Sollen Daten asynchron über eine serielle Schnittstelle ausgetauscht werden, kann auf das I/O Objekt `boost::asio::serial_port` zugegriffen werden. Soll lediglich asynchron auf den Ablauf einer Zeitspanne gewartet werden, kann das I/O Objekt `boost::asio::steady_timer` verwendet werden.
`boost::asio::steady_timer` ist vergleichbar mit einem Wecker. Anstatt in einer blockierenden Funktion warten zu müssen, bis der Wecker klingelt, wird Ihr Programm nach Ablauf der Zeitspanne informiert. Da `boost::asio::steady_timer` lediglich auf den Ablauf einer Zeitspanne wartet, findet über dieses I/O Objekt scheinbar kein Zugriff auf eine externe Ressource statt. Die externe Ressource ist in diesem Fall die Fähigkeit des Betriebssystems, nach Ablauf der Zeitspanne dem Programm Bescheid zu geben, so dass im Programm zum Beispiel kein neuer Thread erstellt und in diesem mit einer blockierenden Funktion gewartet werden muss. Da es sich bei `boost::asio::steady_timer` um ein sehr einfaches I/O Objekt handelt, wird mit diesem in die Entwicklung auf Boost.Asio basierenden Programmen eingestiegen.

> **Anmerkung**
>
> Einige der folgenden Beispiele in diesem Kapitel können aufgrund eines Bugs in Boost.Asio 1.57.0 nicht mit Clang kompiliert werden. Der Bug ist im Ticket 8835 beschrieben. Wenn Sie die Typen aus `std::chrono` mit den entsprechenden Typen aus `boost::chrono` ersetzen, können Sie die Beispiele auch mit Clang übersetzen.

Beispiel 32.1 `boost::asio::steady_timer` in Aktion

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  steady_timer timer{ioservice, std::chrono::seconds{3}};
  timer.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  ioservice.run();
}
```

Im Beispiel 32.1 wird in `main()` ein I/O Serviceobjekt **`ioservice`** erstellt, mit dem das I/O Objekt **`timer`** initialisiert wird. So wie `boost::asio::steady_timer` erwarten typischerweise alle I/O Objekte als ersten Parameter im Konstruktor ein I/O Serviceobjekt. Da **`timer`** einen Wecker darstellt, kann dem Konstruktor von `boost::asio::steady_timer` ein zweiter Parameter übergeben werden, der einen Zeitpunkt oder eine Zeitspanne angibt, nach deren Ablauf der Wecker klingeln soll. Im Beispiel 32.1 wird angegeben, dass der Wecker nach drei Sekunden klingeln soll. Die Zeit beginnt ab der **`timer`**-Definition zu laufen.
Anstatt eine blockierende Funktion aufzurufen, die nach drei Sekunden zurückkehrt, wenn der Wecker klingelt, kann mit Boost.Asio eine asynchrone Operation gestartet werden. Dazu wird die Methode `async_wait()` aufgerufen, der als einziger Parameter ein *Handler* übergeben wird. Ein Handler ist eine Funktion oder ein Funktionsobjekt, das aufgerufen wird, wenn die asynchrone Operation beendet wurde. Im obigen Beispiel wird eine Lambda-Funktion als Handler übergeben.

`async_wait()` kehrt sofort zurück. Anstatt drei Sekunden zu warten, bis der Wecker klingelt, wird nach drei Sekunden die Lambda-Funktion aufgerufen. Das Programm kann nach dem Aufruf von `async_wait()` etwas anderes tun als nur zu warten.
Eine Methode wie `async_wait()` wird als nicht-blockierend bezeichnet. Üblicherweise bieten I/O Objekte auch blockierende Methoden an. So kann zum Beispiel für `boost::asio::steady_timer` die blockierende Methode `wait()` aufgerufen werden. Da diese Methode blockiert, wird ihr keine Funktion übergeben. `wait()` kehrt zu einem bestimmten Zeitpunkt oder nach Ablauf einer Zeitspanne zurück.
Die letzte Anweisung in `main()` im Beispiel 32.1 ist der Aufruf von `run()` für das I/O Serviceobjekt. Dieser Methodenaufruf ist zwingend notwendig, da die betriebssystemeigenen Funktionen die Kontrolle übernehmen und nach drei Sekunden die Lambda-Funktion aufrufen müssen. Erinnern Sie sich, dass es die I/O Services im I/O Serviceobjekt sind, die asynchrone Operationen basierend auf Betriebssystemfunktionen implementieren. Während `async_wait()` eine asynchrone Operation initiiert und sofort zurückkehrt, blockiert `run()`. Der Grund ist, dass viele Betriebssysteme asynchrone Operationen nur über eine blockierende Funktion unterstützen. Warum das in der Praxis üblicherweise kein Problem ist, sehen Sie anhand des folgenden Beispiels.

Beispiel 32.2 Zwei asynchrone Operationen mit `boost::asio::steady_timer`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  steady_timer timer1{ioservice, std::chrono::seconds{3}};
  timer1.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  steady_timer timer2{ioservice, std::chrono::seconds{4}};
  timer2.async_wait([](const boost::system::error_code &ec)
    { std::cout << "4 sec\n"; });

  ioservice.run();
}
```

Im Beispiel 32.2 werden zwei I/O Objekte vom Typ `boost::asio::steady_timer` verwendet. Das erste I/O Objekt repräsentiert einen Wecker, der nach drei Sekunden, das zweite einen Wecker, der nach vier Sekunden klingelt. Nach Ablauf der Zeitspannen werden die beiden Lambda-Funktionen aufgerufen, die an `async_wait()` übergeben wurden.
Auch in diesem Beispiel wird am Ende der Funktion `main()` die Methode `run()` für das einzige I/O Serviceobjekt aufgerufen. Durch diesen Methodenaufruf wird die Steuerung an Betriebssystemfunktionen übergeben, die die asynchrone Datenverarbeitung übernehmen. Mit ihrer Hilfe wird die erste Lambda-Funktion nach drei Sekunden und die zweite Lambda-Funktion nach vier Sekunden aufgerufen.
Es mag auf den ersten Blick verwundern, dass die asynchrone Datenverarbeitung den Aufruf einer blockierenden Methode erfordert. Das ist jedoch insofern kein Problem, als dass das Programm sowieso davor bewahrt werden muss, beendet zu werden. Würde `run()` nicht blockieren, würde die Funktion `main()` beendet werden – und damit das Programm. Für den Fall, dass nicht auf die Rückkehr des Methodenaufrufs gewartet werden und das Programm weiterlaufen soll, muss `run()` lediglich in einem neuen Thread aufgerufen werden.
Dass die Beispielprogramme dennoch beendet werden, liegt daran, dass `run()` zurückkehrt, wenn das I/O Serviceobjekt, für das diese Methode aufgerufen wurde, nichts mehr zu tun hat. Für die obigen Programme bedeutet dies, dass sie dann beendet werden, wenn alle Wecker geklingelt haben. Dann gibt es keine asynchronen Operationen mehr, die auf ihren Abschluss warten.

32.2 Skalierbarkeit und Multithreading

Wenn Sie auf eine Bibliothek wie Boost.Asio zugreifen, entwickeln Sie ein Programm anders als üblicherweise in C++ gewohnt. So geben Sie nicht aktiv die Reihenfolge aller Funktionsaufrufe vor. Die Reihenfolge, in der

Handler ausgeführt werden, hängt von der Reihenfolge ab, in der asynchrone Operationen beendet werden – und die ist nicht unbedingt vorhersehbar. Dies macht es nicht einfacher, die Programmlogik nachzuvollziehen.
Eine Bibliothek wie Boost.Asio wird typischerweise dann eingesetzt, wenn eine höhere Effizienz erreicht werden soll. Ein Programm soll nicht mehr warten, bis eine Operation abgeschlossen wurde, sondern zwischenzeitlich anderes tun können. So können in einem Programm, das auf Boost.Asio basiert, beliebig viele asynchrone Operationen gestartet werden, die alle gleichzeitig ausgeführt werden – denken Sie daran, dass über asynchrone Operationen üblicherweise auf Ressourcen außerhalb des Programms zugegriffen wird. Da diese Ressourcen unterschiedliche Hardware-Komponenten Ihres Computers sein können, können sie unabhängig voneinander arbeiten und Operationen gleichzeitig ausführen.
Skalierbarkeit bezeichnet die Eigenschaft eines Programms, von zusätzlichen Ressourcen zu profitieren, wenn diese zur Verfügung gestellt werden. Mit Boost.Asio kann davon profitiert werden, dass mehrere Operationen auf externen Geräten gleichzeitig zum Code in einem Programm ausgeführt werden können. Werden zusätzlich zu Boost.Asio Threads verwendet, können außerdem mehrere Operationen in einem Programm gleichzeitig auf den verfügbaren Prozessorkernen ausgeführt werden. Boost.Asio mit Threads verbessert die Skalierbarkeit eines Programms, indem von möglichst vielen zur Verfügung stehenden internen und externen Geräten profitiert wird, die in der Lage sind, Operation selbstständig und im Verbund gleichzeitig auszuführen.
Wenn die Methode `run()` für ein Objekt vom Typ `boost::asio::io_service` aufgerufen wird, erfolgt der Aufruf von Handlern im gleichen Thread, in dem `run()` aufgerufen wurde. Verwendet ein Programm mehrere Threads, kann in jedem Thread ein Aufruf von `run()` erfolgen. Das I/O Serviceobjekt wird dann, wenn eine asynchrone Operation abgeschlossen wurde, den Handler in einem dieser Threads ausführen. Sollte eine zweite asynchrone Operation abgeschlossen werden, kann das I/O Serviceobjekt den entsprechenden Handler in einem anderen zur Verfügung stehenden Thread starten. Der Vorteil ist, dass Operationen nicht nur außerhalb des Programms gleichzeitig ausgeführt werden können, sondern auch Handler in einem Programm.

Beispiel 32.3 Zwei Threads für das I/O Serviceobjekt, um Handler zeitgleich auszuführen

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <thread>
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  steady_timer timer1{ioservice, std::chrono::seconds{3}};
  timer1.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  steady_timer timer2{ioservice, std::chrono::seconds{3}};
  timer2.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  std::thread thread1{[&ioservice](){ ioservice.run(); }};
  std::thread thread2{[&ioservice](){ ioservice.run(); }};
  thread1.join();
  thread2.join();
}
```

Beispiel 32.2 wird im Beispiel 32.3 in ein Multithreaded-Programm umgewandelt. In der Funktion `main()` werden mit Hilfe der Klasse `std::thread` zwei Threads erstellt. In beiden Threads wird lediglich die Methode `run()` für das einzige I/O Serviceobjekt aufgerufen. Damit erhält das I/O Serviceobjekt die Möglichkeit, beide Threads zu nutzen, um Handler auszuführen, die bei Abschluss asynchroner Operationen aufgerufen werden müssen.
Im Beispiel 32.3 sollen beide Wecker nach drei Sekunden klingeln. Dadurch, dass zwei Threads zur Verfügung stehen, können beide Lambda-Funktionen zeitgleich ausgeführt werden. Für den Fall, dass der zweite Wecker klingelt und momentan der Handler des ersten Weckers ausgeführt wird, wird der Handler im zweiten zur Verfügung stehenden Thread ausgeführt. Sollte der Handler des ersten Weckers bereits beendet worden sein, steht es dem I/O Serviceobjekt frei, einen der beiden zur Verfügung stehenden Threads für den zweiten Handler zu ver-

wenden.
Beachten Sie, dass der Einsatz von Threads nicht immer sinnvoll ist. Wenn Sie Beispiel 32.3 ausführen, kann es sein, dass die beiden Meldungen auf der Standardausgabe nicht nacheinander erscheinen, sondern gemischt werden. Beide Handler, die möglicherweise gleichzeitig in zwei Threads ausgeführt werden, greifen mit **`std::cout`** auf eine einzige Ressource zu und müssen sie sich teilen. Um sicherzustellen, dass eine Meldung vollständig ausgegeben wird, ohne dass gleichzeitig ein anderer Thread eine Meldung auf die Standardausgabe auszugeben versucht, müsste der Zugriff synchronisiert werden. Der Vorteil von Threads ist dahin, wenn Handler nicht unabhängig voneinander laufen können, sondern synchronisiert werden müssen.

Beispiel 32.4 Je ein Thread für zwei I/O Serviceobjekte, um Handler zeitgleich auszuführen

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <thread>
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice1;
  io_service ioservice2;

  steady_timer timer1{ioservice1, std::chrono::seconds{3}};
  timer1.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  steady_timer timer2{ioservice2, std::chrono::seconds{3}};
  timer2.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  std::thread thread1{[&ioservice1](){ ioservice1.run(); }};
  std::thread thread2{[&ioservice2](){ ioservice2.run(); }};
  thread1.join();
  thread2.join();
}
```

Der mehrfache Aufruf von `run()` eines I/O Serviceobjekts ist die empfohlene Vorgehensweise, um ein Programm, das auf Boost.Asio basiert, skalierbar zu machen. Sie können jedoch auch, anstatt mehrere Threads an ein I/O Serviceobjekt zu binden, mehrere I/O Serviceobjekte instanziieren.
Im Beispiel 32.4 werden neben zwei Weckern vom Typ `boost::asio::steady_timer` zwei I/O Serviceobjekte verwendet. Das Programm basiert auf zwei Threads, wobei jeweils ein Thread an ein I/O Serviceobjekt gebunden ist. Die beiden I/O Objekte **`timer1`** und **`timer2`** sind nicht mehr an das gleiche I/O Serviceobjekt gebunden, sondern an unterschiedliche.
Beispiel 32.4 funktioniert wie das vorherige, das lediglich ein I/O Serviceobjekt verwendet. Wann es von Vorteil ist, mehr als ein I/O Serviceobjekt zu verwenden, lässt sich nicht allgemein beantworten. Da `boost::asio::io_service` die Betriebssystemschnittstelle darstellt, hängt es von dieser ab, ob und wann mehr als eine Instanz von Vorteil ist. Unter Windows verbirgt sich hinter `boost::asio::io_service` üblicherweise IOCP, unter Linux `epoll()`. Mehrere `boost::asio::io_service`-Instanzen bedeuten demnach, dass mehrere I/O Completion Ports verwendet werden oder mehrfach `epoll()` aufgerufen wird. Ob dies besser ist als alle asynchronen Operationen über einen I/O Completion Port oder einen Aufruf von `epoll()` zu verwalten, hängt vom Einzelfall ab.

32.3 Netzwerkprogrammierung

Auch wenn Boost.Asio eine Bibliothek ist, mit der beliebige Daten asynchron verarbeitet werden können, wird sie in der Praxis häufig zur Netzwerkprogrammierung eingesetzt. Boost.Asio unterstützte Netzwerkfunktionen zuerst, bevor im Laufe der Zeit neue I/O Objekte hinzukamen. Netzwerkfunktionen sind insofern ein gutes Beispiel für die asynchrone Datenverarbeitung, da Netzwerkverbindungen externe Ressourcen sind und eine Datenübertragung über Netzwerke unter Umständen einige Zeit dauern kann. Ergebnisse wie Empfangsbestätigungen

oder Fehlermeldungen liegen nicht so schnell vor wie der Code der Funktionen, die Sie zum Datenversand oder -empfang in Ihrem Programm aufrufen, ausgeführt werden kann.
Boost.Asio bietet zahlreiche I/O Objekte, um Netzwerkanwendungen zu entwickeln. Im Beispiel 32.5 lernen Sie die Klasse `boost::asio::ip::tcp::socket` kennen, über die Sie eine Verbindung zu einem anderen Computer aufbauen können. Das Beispiel sendet einen HTTP-Request an einen Webserver, um die Homepage herunterzuladen.

Beispiel 32.5 Ein Webclient mit `boost::asio::ip::tcp::socket`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/buffer.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <array>
#include <string>
#include <iostream>

using namespace boost::asio;
using namespace boost::asio::ip;

io_service ioservice;
tcp::resolver resolv{ioservice};
tcp::socket tcp_socket{ioservice};
std::array<char, 4096> bytes;

void read_handler(const boost::system::error_code &ec,
  std::size_t bytes_transferred)
{
  if (!ec)
  {
    std::cout.write(bytes.data(), bytes_transferred);
    tcp_socket.async_read_some(buffer(bytes), read_handler);
  }
}

void connect_handler(const boost::system::error_code &ec)
{
  if (!ec)
  {
    std::string r =
      "GET / HTTP/1.1\r\nHost: theboostcpplibraries.com\r\n\r\n";
    write(tcp_socket, buffer(r));
    tcp_socket.async_read_some(buffer(bytes), read_handler);
  }
}

void resolve_handler(const boost::system::error_code &ec,
  tcp::resolver::iterator it)
{
  if (!ec)
    tcp_socket.async_connect(*it, connect_handler);
}

int main()
{
  tcp::resolver::query q{"theboostcpplibraries.com", "80"};
  resolv.async_resolve(q, resolve_handler);
  ioservice.run();
}
```

Im Beispiel 32.5 werden drei Handler verwendet: Die Funktionen `connect_handler()` und `read_handler()` werden aufgerufen, wenn die Verbindung erstellt wird und Daten empfangen werden. `resolve_handler()` wird zur Namensauflösung verwendet.
Weil der Empfang von Daten eine erfolgreiche Verbindungsaufnahme voraussetzt und eine Verbindungsaufnahme eine erfolgreiche Namensauflösung, werden verschiedene asynchrone Operationen in Handlern gestartet. So

wird in der Funktion `resolve_handler()` auf das I/O Objekt **`tcp_socket`** zugegriffen, um mit Hilfe der aufgelösten Adresse, die über den Iterator **`it`** zur Verfügung steht, eine Verbindung aufzubauen. In der Funktion `connect_handler()` wird auf **`tcp_socket`** zugegriffen, um einen HTTP-Request zu senden und den Datenempfang zu starten. Weil es sich bei allen Funktionen um asynchrone Operationen handelt, werden jeweils die Namen der Handler als Parameter weitergegeben. Je nach Funktion sind zusätzliche Parameter notwendig wie beispielsweise der Iterator **`it`**, der auf die aufgelöste Adresse zeigt, oder das Array **`bytes`**, in dem empfangene Daten gespeichert werden.
Wenn Sie das Beispiel ausführen, wird in der Funktion `main()` ein Objekt **`q`** vom Typ `boost::asio::ip::tcp::resolver::query` erstellt. Es handelt sich hierbei um den Typ für Anfragen an den Namensauflöser. Der Namensauflöser ist ein I/O Objekt vom Typ `boost::asio::ip::tcp::resolver`. Indem **`q`** an `async_resolve()` übergeben wird, wird eine asynchrone Operation zur Namensauflösung gestartet. Im Beispiel soll der Name theboostcpplibraries.com aufgelöst werden. Nachdem die asynchrone Operation gestartet wurde, wird `run()` für das I/O Serviceobjekt aufgerufen, um die Kontrolle über die asynchronen Operationen ans Betriebssystem zu übergeben.
Wurde der Name aufgelöst, wird der Handler `resolve_handler()` aufgerufen. In diesem wird überprüft, ob die Namensauflösung erfolgreich war. Ist sie das, ist das Objekt **`ec`**, das Fehlerarten repräsentiert, auf 0 gesetzt. Nur in diesem Fall wird auf den Socket zugegriffen und der Verbindungsaufbau initiiert. Die Adresse des Servers, zu dem die Verbindung aufgebaut werden soll, steht über den zweiten Funktionsparameter vom Typ `boost::asio::ip::tcp::resolver::iterator` zur Verfügung. Dies ist das Ergebnis der asynchronen Namensauflösung.
Dem Aufruf von `async_connect()` folgt ein Aufruf des Handlers `connect_handler()`. Dort wird ebenfalls ein Objekt **`ec`** ausgewertet, um zu überprüfen, ob der Verbindungsaufbau erfolgreich war. Ist dies der Fall, wird die Methode `async_read_some()` für den Socket aufgerufen. Mit diesem Methodenaufruf beginnt der Lesevorgang über die nun bestehende Verbindung. Empfangene Daten werden im Array **`bytes`** gespeichert, das als erster Parameter an `async_read_some()` übergeben wird.
Die Funktion `read_handler()` wird aufgerufen, wenn ein oder mehr Bytes empfangen und in **`bytes`** gespeichert wurden. Der Parameter **`bytes_transferred`** vom Typ std::size_t gibt an, wie viele Bytes empfangen wurden. Wie üblich sollte auch in diesem Handler zuerst der Parameter **`ec`** ausgewertet werden, um zu überprüfen, ob möglicherweise ein Empfangsfehler vorliegt. Nur wenn dies nicht der Fall ist, werden die empfangenen Daten auf die Standardausgabe ausgegeben.
Beachten Sie, dass innerhalb des Handlers `read_handler()` nach der Datenausgabe über **`std::cout`** `async_read_some()` erneut für den Socket aufgerufen wird. Das ist notwendig, da nicht davon ausgegangen werden kann, dass die gesamte Homepage über eine einzige asynchrone Operation empfangen werden kann und vollständig in **`bytes`** vorliegt. Der wiederholte Aufruf von `async_read_some()` gefolgt von einem wiederholten Aufruf des Handlers `read_handler()` endet erst dann, wenn die Verbindung unterbrochen wird. Dies geschieht, wenn der Webserver die Homepage komplett versendet hat. In diesem Fall wird im Handler `read_handler()` ein Fehler gemeldet, so dass keine Datenausgabe über **`std::cout`** erfolgt und `async_read()` für den Socket nicht mehr aufgerufen wird. Da es keine ausstehenden asynchronen Operationen mehr gibt, endet das Programm.

Beispiel 32.6 Ein Zeitserver mit `boost::asio::ip::tcp::acceptor`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/buffer.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <string>
#include <ctime>

using namespace boost::asio;
using namespace boost::asio::ip;

io_service ioservice;
tcp::endpoint tcp_endpoint{tcp::v4(), 2014};
tcp::acceptor tcp_acceptor{ioservice, tcp_endpoint};
tcp::socket tcp_socket{ioservice};
std::string data;

void write_handler(const boost::system::error_code &ec,
  std::size_t bytes_transferred)
{
  if (!ec)
    tcp_socket.shutdown(tcp::socket::shutdown_send);
```

```
}

void accept_handler(const boost::system::error_code &ec)
{
  if (!ec)
  {
    std::time_t now = std::time(nullptr);
    data = std::ctime(&now);
    async_write(tcp_socket, buffer(data), write_handler);
  }
}

int main()
{
  tcp_acceptor.listen();
  tcp_acceptor.async_accept(tcp_socket, accept_handler);
  ioservice.run();
}
```

Beispiel 32.6 ist ein Zeitserver. Sie können mit einem Telnet-Client eine Verbindung aufbauen und die aktuelle Zeit erhalten. Anschließend wird der Zeitserver beendet.
Das I/O Objekt `boost::asio::ip::tcp::acceptor` wird verwendet, um auf einen Verbindungsaufbau ausgehend von einem anderen Programm zu warten. Dazu muss das entsprechende Objekt, im Beispiel 32.6 **`tcp_acceptor`**, derart initialisiert werden, dass es weiß, über welches Protokoll und über welchen Port ein möglicher Verbindungsaufbau erfolgt. Mit **`tcp_endpoint`** vom Typ `boost::asio::ip::tcp::endpoint` wird angegeben, dass der Acceptor auf Port 2014 auf eingehende Verbindungen vom Typ des Internet-Protokolls 4 warten soll.
Nachdem der Acceptor initialisiert wurde, wird in der Funktion `main()` `listen()` aufgerufen, um den Acceptor in den Empfangsmodus zu setzen. Anschließend wird mit `async_accept()` auf die erste Verbindungsaufnahme gewartet. Dazu muss ein Socket als erster Parameter an `async_accept()` übergeben werden, über den der Datenversand und -empfang erfolgen soll.
Nimmt ein anderes Programm Verbindung auf, wird die Funktion `accept_handler()` aufgerufen. War der Verbindungsaufbau erfolgreich, wird die aktuelle Zeit ermittelt und die freistehende Funktion `boost::asio::async_write()` aufgerufen. Diese Funktion wird verwendet, um sämtliche Daten in **`data`** über den Socket zu verschicken. Die Klasse `boost::asio::ip::tcp::socket` bietet auch eine Methode `async_write_some()` an. Bei dieser Methode wird ein Handler immer dann aufgerufen, wenn mindestens ein Byte gesendet wurde. Es müsste im Handler berechnet werden, wie viele Bytes noch gesendet werden müssen. Diese müssten über einen wiederholten Aufruf von `async_write_some()` verschickt werden. Berechnungen und wiederholte Aufrufe von `async_write_some()` lassen sich vermeiden, wenn `boost::asio::async_write()` verwendet wird, da diese asynchrone Operation erst als abgeschlossen gilt, wenn alle Daten in **`data`** gesendet wurden.
Nach dem Versand aller Daten wird die Funktion `write_handler()` aufgerufen. Diese Funktion greift auf den Socket zu und ruft `shutdown()` auf. Mit dem übergebenen Parameter **`boost::asio::ip::tcp::socket::shutdown_send`** wird angegeben, dass der Datenversand über den Socket abgeschlossen ist. Da es keine ausstehenden asynchronen Operationen gibt, endet Beispiel 32.6.
Beachten Sie, dass **`data`** keine lokale Variable ist. Obwohl **`data`** ausschließlich in `accept_handler()` verwendet wird, darf es sich bei diesem String nicht um eine lokale Variable handeln. Die Übergabe von **`data`** über `boost::asio::buffer()` an `boost::asio::async_write()` erfolgt per Referenz. Wenn `boost::asio::async_write()` und anschließend `accept_handler()` zurückkehrt, ist die asynchrone Operation initiiert, aber nicht beendet. **`data`** muss jedoch solange existieren, bis die asynchrone Operation beendet ist und die Variable nicht mehr benötigt wird. Indem **`data`** als globale Variable definiert ist, ist sichergestellt, dass sich der Gültigkeitsbereich der Variablen über die gesamte asynchrone Operation erstreckt.

32.4 Coroutinen

Seit der Version 1.54.0 unterstützt Boost.Asio Coroutinen. Während Sie die Bibliothek Boost.Coroutine direkt verwenden könnten, vereinfacht die explizite Unterstützung in Boost.Asio den Einsatz von Coroutinen.
Mit Coroutinen ist es möglich, auf Boost.Asio basierende Programme so zu strukturieren, dass Funktionsaufrufe die eigentliche Programmlogik widerspiegeln. Asynchrone Operationen verursachen keinen Bruch mehr, weil

Handler definiert werden müssen, die Code beinhalten, der ausgeführt werden soll, wenn eine asynchrone Operation endet. Anstatt zahlreiche aufeinander zugreifende Handler zu definieren, erhält der Code seine ursprüngliche sequentielle Struktur wieder.

Beispiel 32.7 Coroutinen mit Boost.Asio

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/spawn.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/buffer.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <list>
#include <string>
#include <ctime>

using namespace boost::asio;
using namespace boost::asio::ip;

io_service ioservice;
tcp::endpoint tcp_endpoint{tcp::v4(), 2014};
tcp::acceptor tcp_acceptor{ioservice, tcp_endpoint};
std::list<tcp::socket> tcp_sockets;

void do_write(tcp::socket &tcp_socket, yield_context yield)
{
  std::time_t now = std::time(nullptr);
  std::string data = std::ctime(&now);
  async_write(tcp_socket, buffer(data), yield);
  tcp_socket.shutdown(tcp::socket::shutdown_send);
}

void do_accept(yield_context yield)
{
  for (int i = 0; i < 2; ++i)
  {
    tcp_sockets.emplace_back(ioservice);
    tcp_acceptor.async_accept(tcp_sockets.back(), yield);
    spawn(ioservice, [](yield_context yield)
      { do_write(tcp_sockets.back(), yield); });
  }
}

int main()
{
  tcp_acceptor.listen();
  spawn(ioservice, do_accept);
  ioservice.run();
}
```

Die wichtigste Funktion für den Einsatz von Coroutinen mit Boost.Asio ist `boost::asio::spawn()`. Dieser Funktion muss als erster Parameter ein I/O Serviceobjekt übergeben werden. Der zweite Parameter ist eine Funktion, die als Coroutine dienen soll. Diese Funktion muss als einzigen Parameter ein Objekt vom Typ `boost::asio::yield_context` erwarten und darf keinen Rückgabewert haben. Im Beispiel 32.7 werden `do_accept()` und `do_write()` als Coroutinen verwendet. Ist die Signatur wie im Fall von `do_write()` verschieden, muss ein Adapter verwendet werden – zum Beispiel `std::bind` oder wie hier eine Lambda-Funktion. Ein Objekt vom Typ `boost::asio::yield_context` kann anstatt eines Handlers an asynchrone Funktionen übergeben werden. So wird innerhalb von `do_accept()` der Parameter **`yield`** an `async_accept()` übergeben. In `do_write()` wird **`yield`** an `async_write()` übergeben. Diese Methoden rufen demnach keinen Handler auf, wenn die asynchronen Operationen beendet wurden. Stattdessen stellen sie den Kontext wieder her, in dem sie aufgerufen wurden. Dies bedeutet, dass das Programm nach Abschluß einer asynchronen Operation dort fortsetzt, wo die asynchrone Operation gestartet worden war.
`do_accept()` enthält eine `for`-Schleife. In dieser Schleife wird jeweils ein neuer Socket an `async_accept()` übergeben, um eine neue Verbindung zu akzeptieren. Hat ein Client Verbindung aufgenommen, wird über `boost::asio::spawn()` die Coroutine `do_write()` aufgerufen, um die aktuelle Uhrzeit an den Client zu senden.

Anhand des Schleifenkopfs ist erkennbar, dass das Programm zwei Verbindungen annehmen und die aktuelle Uhrzeit an zwei Clients senden kann, bevor es endet. Weil das Beispiel auf Coroutinen basiert, kann die wiederholte Ausführung einer Operation in einer herkömmlichen `for`-Schleife erfolgen. Dies kann das Verständnis des Codes erleichtern, da nicht mehr die potentielle Reihenfolge von Handler-Aufrufen durchdacht werden muss, um herauszufinden, wann die letzte asynchrone Operation ausgeführt wird. Soll der Zeitserver mehr als zwei Clients bedienen können, muss lediglich die `for`-Schleife angepasst werden.

32.5 Plattformspezifische I/O Objekte

Alle bisherigen Beispiele in diesem Kapitel sind plattformunabhängig. So stehen I/O Objekte wie `boost::asio::steady_timer` oder `boost::asio::ip::tcp::socket` auf allen Plattformen zur Verfügung. Boost.Asio bietet jedoch auch plattformspezifische I/O Objekte an, über die Sie asynchrone Operationen anstoßen können, wie sie beispielsweise nur unter Windows oder nur unter POSIX-Betriebssystemen möglich sind.

Beispiel 32.8 `boost::asio::windows::object_handle` in Aktion

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/windows/object_handle.hpp>
#include <boost/system/error_code.hpp>
#include <iostream>
#include <Windows.h>

using namespace boost::asio;
using namespace boost::system;

int main()
{
  io_service ioservice;

  HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED, NULL);

  char buffer[1024];
  DWORD transferred;
  OVERLAPPED overlapped;
  ZeroMemory(&overlapped, sizeof(overlapped));
  overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
    FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

  windows::object_handle obj_handle{ioservice, overlapped.hEvent};
  obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
    if (!ec)
    {
      DWORD transferred;
      GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
        FALSE);
      auto notification = reinterpret_cast<FILE_NOTIFY_INFORMATION*>(buffer);
      std::wcout << notification->Action << '\n';
      std::streamsize size = notification->FileNameLength / sizeof(wchar_t);
      std::wcout.write(notification->FileName, size);
    }
  });

  ioservice.run();
}
```

Im Beispiel 32.8 wird das I/O Objekt `boost::asio::windows::object_handle` verwendet, das ausschließlich unter Windows zur Verfügung steht. `boost::asio::windows::object_handle` ermöglicht es, asynchrone Operationen für Objekt-Handles anzustoßen, die an die Windows-Funktion `RegisterWaitForSingleObject()` übergeben werden können. `boost::asio::windows::object_handle` basiert auf `RegisterWaitFo`

rSingleObject(). Mit async_wait() bietet das I/O Objekt die Möglichkeit an, asynchron auf eine Veränderung eines Objekt-Handles zu warten.
Im obigen Beispiel wird **obj_handle** vom Typ boost::asio::windows::object_handle mit einem Handle initialisiert, der mit der Windows-Funktion CreateEvent() erstellt wurde. Der Handle ist Teil einer OVERLAPPED-Struktur, deren Adresse an die Windows-Funktion ReadDirectoryChangesW() übergeben wird. OVERLAPPED-Strukturen werden unter Windows verwendet, um asynchrone Operationen anzustoßen.
Mit ReadDirectoryChangesW() kann ein Verzeichnis auf Veränderungen überwacht werden. Damit die Funktion als asynchrone Operation ausgeführt wird, muss eine OVERLAPPED-Struktur übergeben werden. Damit der Abschluß der asynchronen Operation über Boost.Asio gemeldet wird, wird ein Event-Handle in der OVERLAPPED-Struktur gesetzt, bevor diese an ReadDirectoryChangesW() übergeben wird. Indem dieser Handle anschließend an **obj_handle** übergeben und async_wait() aufgerufen wird, kann im Handler auf eine Veränderung im Verzeichnis reagiert werden.
Wenn Sie obiges Beispiel ausführen und im gleichen Verzeichnis, in dem Sie das Beispiel ausführen, eine neue Datei erstellen, wird für diese Veränderung im Verzeichnis eine entsprechende Meldung auf die Standardausgabe ausgegeben.

Beispiel 32.9 boost::asio::windows::overlapped_ptr in Aktion

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/windows/overlapped_ptr.hpp>
#include <boost/system/error_code.hpp>
#include <iostream>
#include <Windows.h>

using namespace boost::asio;
using namespace boost::system;

int main()
{
  io_service ioservice;

  HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED, NULL);

  error_code ec;
  auto &io_service_impl = use_service<detail::io_service_impl>(ioservice);
  io_service_impl.register_handle(file_handle, ec);

  char buffer[1024];
  auto handler = [&buffer](const error_code &ec, std::size_t) {
    if (!ec)
    {
      auto notification =
        reinterpret_cast<FILE_NOTIFY_INFORMATION*>(buffer);
      std::wcout << notification->Action << '\n';
      std::streamsize size = notification->FileNameLength / sizeof(wchar_t);
      std::wcout.write(notification->FileName, size);
    }
  };
  windows::overlapped_ptr overlapped{ioservice, handler};
  DWORD transferred;
  BOOL ok = ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer),
    FALSE, FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, overlapped.get(),
    NULL);
  int last_error = GetLastError();
  if (!ok && last_error != ERROR_IO_PENDING)
  {
    error_code ec{last_error, error::get_system_category()};
    overlapped.complete(ec, 0);
  }
  else
  {
    overlapped.release();
```

```
  }

  ioservice.run();
}
```

Im Beispiel 32.9 wird wie im vorherigen mit `ReadDirectoryChangesW()` ein Verzeichnis überwacht. Diesmal ist der asynchrone Aufruf von `ReadDirectoryChangesW()` nicht mit einem Event-Handle verknüpft. Das Beispiel verwendet eine Klasse `boost::asio::windows::overlapped_ptr`, die intern eine `OVERLAPPED`-Struktur verwendet. Über `get` wird ein Zeiger auf die interne `OVERLAPPED`-Struktur erhalten. Dieser wird im Beispiel an `ReadDirectoryChangesW()` übergeben.
Bei `boost::asio::windows::overlapped_ptr` handelt es sich um ein I/O Objekt, das keine Methode anbietet, um eine asynchrone Operation zu starten. Die asynchrone Operation wird gestartet, indem ein Zeiger auf die interne `OVERLAPPED`-Struktur an eine entsprechende Windows-Funktion übergeben wird. Der Konstruktor von `boost::asio::windows::overlapped_ptr` erwartet neben einer Referenz auf ein I/O Serviceobjekt einen Handler, der aufgerufen wird, wenn die asynchrone Operation beendet wurde.
Beachten Sie, dass im Beispiel mit `boost::asio::use_service()` eine Referenz auf einen Service im I/O Serviceobjekt **`ioservice`** erhalten wird. `boost::asio::use_service()` ist eine Template-Funktion, der der Typ des Services als Template-Parameter übergeben werden muss. Im Beispiel wird über den Typ boost::asio::detail::io_service_
Zugriff auf den I/O Service erhalten, der die betriebssystemnächste Schnittstelle im I/O Serviceobjekt ist. Unter Windows greift boost::asio::detail::io_service_impl auf IOCP zu, unter Linux zum Beispiel auf `epoll()`. boost::asio::detail::io_service_impl ist eine Typdefinition, die unter Windows auf `boost::asio::detail::win_iocp_io_service` und unter Linux auf `boost::asio::detail::task_io_service` gesetzt sein kann. `boost::asio::detail::win_iocp_io_service` bietet eine Methode `register_handle()` an, um einen Handle mit dem IOCP-Handle zu verknüpfen. `register_handle()` ruft die Windows-Funktion `CreateIoCompletionPort()` auf. Dies ist nötig, damit das Beispiel richtig funktioniert. Der Handle, der von `CreateFileA()` zurückgegeben wird, darf erst nach einer derartigen Verknüpfung über **`overlapped`** an `ReadDirectoryChangesW()` übergeben werden.
Beachten Sie außerdem, dass für **`overlapped`** nach dem Aufruf von `ReadDirectoryChangesW()` `complete()` oder `release()` aufgerufen werden muss. Im Beispiel wird überprüft, ob `ReadDirectoryChangesW()` fehlgeschlagen ist und die asynchrone Operation womöglich schon beendet ist. In diesem Fall wird `complete()` aufgerufen, um die asynchrone Operation für Boost.Asio zu beenden. Die Parameter, die an `complete()` übergeben werden, werden an den Handler weitergereicht.
War der Aufruf von `ReadDirectoryChangesW()` erfolgreich, wird `release()` aufgerufen. In diesem Fall gilt die asynchrone Operation als schwebend und wird erst beendet, wenn die asynchrone Windows-Operation, die mit `ReadDirectoryChangesW()` eingeleitet wurde, beendet ist.

Beispiel 32.10 `boost::asio::posix::stream_descriptor` in Aktion

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/posix/stream_descriptor.hpp>
#include <boost/asio/write.hpp>
#include <boost/system/error_code.hpp>
#include <iostream>
#include <unistd.h>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  posix::stream_descriptor stream{ioservice, STDOUT_FILENO};
  auto handler = [](const boost::system::error_code&, std::size_t) {
    std::cout << ", world!\n";
  };
  async_write(stream, buffer("Hello"), handler);

  ioservice.run();
}
```

Beispiel 32.10 stellt ein I/O Objekt für POSIX-Betriebssysteme vor. `boost::asio::posix::stream_descriptor` kann mit einem File Descriptor initialisiert werden, um eine asynchrone Operation für diesen zu starten.

Im Beispiel wird **`stream`** mit dem File Descriptor `STDOUT_FILENO` verknüpft, um einen String asynchron in die Standardausgabe zu schreiben.

Kapitel 33

Boost.Interprocess

Unter Interprozesskommunikation versteht man Mechanismen, die den Datenaustausch zwischen Prozessen ermöglichen, die auf dem gleichen Computer laufen. Es geht explizit nicht um einen Datenaustausch über Netzwerke. Wenn Sie Daten zwischen Prozessen austauschen möchten, die auf unterschiedlichen Computern laufen und über ein Netzwerk verbunden sind, ist Boost.Asio die richtige Bibliothek.
In diesem Kapitel wird Ihnen die Bibliothek Boost.Interprocess vorgestellt. Diese Bibliothek bietet zahlreiche Klassen an, die betriebssystemspezifische Schnittstellen zur Interprozesskommunikation abstrahieren. Obwohl Konzepte zur Interprozesskommunikation auf verschiedenen Betriebssystemen ähnlich sind, können die Schnittstellen sehr unterschiedlich sein. Mit Boost.Interprocess wird der plattformunabhängige Zugriff auf Funktionen zur Interprozesskommunikation möglich.
Während Sie zum Datenaustausch zwischen Prozessen auch Boost.Asio verwenden können – also auch dann, wenn die Prozesse auf dem gleichen Computer laufen – ist im Allgemeinen die Performance mit Boost.Interprocess besser. Denn Boost.Interprocess greift auf genau die Betriebssystemschnittstellen zu, die für den Datenaustausch zwischen Prozessen auf einem Computer optimiert sind. Boost.Interprocess sollte daher die erste Wahl sein, wenn kein Datenaustausch über Netzwerke erfolgen muss.

33.1 Shared Memory

Shared Memory ist üblicherweise die schnellste Form der Interprozesskommunikation. Dabei wird ein Speicherbereich gleichzeitig mehreren Prozessen zur Verfügung gestellt. So kann ein Prozess Daten in diesen Speicherbereich ablegen, und ein anderer Prozess kann die abgelegten Daten aus dem Speicherbereich lesen.
Boost.Interprocess bietet eine Klasse `boost::interprocess::shared_memory_object` an, die einen derartigen Speicherbereich repräsentiert. Um diese Klasse nutzen zu können, muss die Headerdatei `boost/interprocess/shared_memory_object.hpp` eingebunden werden.

Beispiel 33.1 Shared Memory erstellen

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object shdmem{open_or_create, "Boost", read_write};
  shdmem.truncate(1024);
  std::cout << shdmem.get_name() << '\n';
  offset_t size;
  if (shdmem.get_size(size))
    std::cout << size << '\n';
}
```

Der Konstruktor von `boost::interprocess::shared_memory_object` erwartet drei Parameter. Der erste Parameter gibt an, ob der Shared Memory erstellt oder nur geöffnet werden soll. Im Beispiel 33.1 ist `boost::interprocess::open_or_create` angegeben, was bedeutet, dass der Shared Memory geöffnet wird, wenn er bereits existiert, anderfalls neu erstellt wird.

Das Öffnen setzt voraus, dass auf einen Shared Memory zugegriffen werden kann, der bereits erstellt wurde. Um Speicherbereiche identifizieren zu können, werden ihnen Namen gegeben. Der zweite Parameter, der dem Konstruktor von `boost::interprocess::shared_memory_object` übergeben wird, gibt den Namen an.
Der dritte und letzte Parameter legt fest, wie ein Prozess auf den Shared Memory zugreifen soll. Beispiel 33.1 darf den Shared Memory sowohl lesen als auch schreiben, da `boost::interprocess::read_write` angegeben ist.
Nachdem ein Objekt vom Typ `boost::interprocess::shared_memory_object` erstellt wurde, existiert ein entsprechender Shared Memory. Der Shared Memory ist jedoch von Beginn an 0 Bytes groß. Um den Shared Memory nutzen zu können, muss `truncate()` aufgerufen werden. Dieser Methode wird die Größe in Bytes übergeben, auf die der Shared Memory anwachsen soll. Für Beispiel 33.1 bedeutet dies, dass der Shared Memory Platz für 1024 Bytes bietet.
Beachten Sie, dass Sie `truncate()` nur aufrufen dürfen, wenn Sie den Shared Memory mit `boost::interprocess::read_write` geöffnet haben. Andernfalls wird eine Ausnahme vom Typ `boost::interprocess::interprocess_exception` geworfen.
Sie können `truncate()` mehrfach aufrufen, um die Größe des Shared Memory anzupassen.
Wenn Sie so wie im Beispiel 33.1 einen Shared Memory erstellt haben, können Sie mit Methoden wie `get_name()` und `get_size()` den Namen und die Größe abfragen.
Da Sie den Shared Memory erstellt haben, um über diesen Speicherbereich Daten mit anderen Prozessen auszutauschen, müssen Sie in irgendeiner Weise auf die 1024 Bytes zugreifen. Dazu müssen Sie den Shared Memory in den Speicherbereich eines Prozesses abbilden. Dies erfolgt mit Hilfe der Klasse `boost::interprocess::mapped_region`.
Wenn Sie sich wundern, warum zwei Klassen eingesetzt werden, um auf einen Shared Memory zuzugreifen: Die Klasse `boost::interprocess::mapped_region` kann auch andere Objekte in den Speicherbereich eines Prozesses abbilden. Sie wird nicht nur im Zusammenhang mit `boost::interprocess::shared_memory_object` verwendet.

Beispiel 33.2 Shared Memory in den Speicherbereich eines Prozesses abbilden

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object shdmem{open_or_create, "Boost", read_write};
  shdmem.truncate(1024);
  mapped_region region{shdmem, read_write};
  std::cout << std::hex << region.get_address() << '\n';
  std::cout << std::dec << region.get_size() << '\n';
  mapped_region region2{shdmem, read_only};
  std::cout << std::hex << region2.get_address() << '\n';
  std::cout << std::dec << region2.get_size() << '\n';
}
```

Um die Klasse `boost::interprocess::mapped_region` verwenden zu können, müssen Sie die Headerdatei `boost/interprocess/mapped_region.hpp` einbinden. Dem Konstruktor dieser Klasse müssen Sie als ersten Parameter ein Objekt vom Typ `boost::interprocess::shared_memory_object` übergeben. Der zweite Parameter legt fest, ob nur lesend oder auch schreibend auf den Speicherbereich zugegriffen werden kann.
Im Beispiel 33.2 werden zwei Objekte vom Typ `boost::interprocess::mapped_region` erstellt: Der Shared Memory namens Boost wird zweimal in den Speicherbereich des Prozesses abgebildet. Das Programm gibt über den Aufruf der Methoden `get_address()` und `get_size()` die Adresse und die Größe des abgebildeten Speicherbereichs aus. Während `get_size()` in beiden Fällen den gleichen Wert zurückgibt, nämlich 1024, ist der Rückgabewert der beiden Aufrufe von `get_address()` verschieden.
Im Beispiel 33.3 wird auf die abgebildeten Speicherbereiche zugegriffen, um zum Test eine Zahl zu speichern und zu lesen.

Beispiel 33.3 Eine Zahl in den Shared Memory schreiben und wieder lesen

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>
```

```
using namespace boost::interprocess;

int main()
{
  shared_memory_object shdmem{open_or_create, "Boost", read_write};
  shdmem.truncate(1024);
  mapped_region region{shdmem, read_write};
  int *i1 = static_cast<int*>(region.get_address());
  *i1 = 99;
  mapped_region region2{shdmem, read_only};
  int *i2 = static_cast<int*>(region2.get_address());
  std::cout << *i2 << '\n';
}
```

Die Zahl 99 wird über **`region`** an den Anfang des 1024 Bytes großen Shared Memory geschrieben. Anschließend wird auf den Anfang des Speicherbereichs **`region2`** zugegriffen und eine Zahl auf die Standardausgabe ausgegeben. Obwohl **`region`** und **`region2`** zwei unterschiedliche Speicherbereiche im Prozess darstellen – deswegen war der Rückgabewert von `get_address()` im vorherigen Beispiel verschieden – gibt das Beispiel `99` aus. Denn sowohl **`region`** als auch **`region2`** greifen auf den gleichen Shared Memory zu.

Beispiel 33.4 Shared Memory löschen

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  bool removed = shared_memory_object::remove("Boost");
  std::cout << std::boolalpha << removed << '\n';
}
```

Wenn Sie einen Shared Memory löschen möchten, greifen Sie auf die statische Methode `remove()` zu, die von der Klasse `boost::interprocess::shared_memory_object` zur Verfügung gestellt wird. Sie müssen ihr wie im Beispiel 33.4 lediglich den Namen des Shared Memory übergeben, der gelöscht werden soll.
Boost.Interprocess unterstützt in gewisser Weise das RAII-Idiom. So können Sie die Klasse `boost::interprocess::remove_shared_memory_on_destroy` verwenden, indem Sie dem Konstruktor den Namen eines existierenden Shared Memory übergeben. Wird das Objekt vom Typ dieser Klasse gelöscht, wird der Shared Memory automatisch im Destruktor freigegeben.
Beachten Sie, dass der Konstruktor von `boost::interprocess::remove_shared_memory_on_destroy` keinen Shared Memory erstellt oder öffnet. Die Klasse ist daher kein typischer Vertreter des RAII-Idioms.
Wenn Sie `remove()` nicht aufrufen, bleibt der Shared Memory bestehen, auch wenn Ihr Programm endet. Es hängt vom Betriebssystem ab, ob und wann der Shared Memory gelöscht wird. Windows und viele Unix-Betriebssysteme inklusive Linux löschen einen Shared Memory automatisch, wenn das System neugestartet wird.
Unter Windows gibt es eine besondere Art von Shared Memory, der automatisch gelöscht wird, wenn der letzte Prozess beendet wurde, der den Shared Memory verwendet hat. Sie können diesen Shared Memory verwenden, indem Sie auf die Klasse `boost::interprocess::windows_shared_memory` zugreifen. Diese Klasse ist in der Headerdatei `boost/interprocess/windows_shared_memory.hpp` definiert. Sehen Sie sich dazu Beispiel 33.5 an.

Beispiel 33.5 Windows-spezifischen Shared Memory verwenden

```
#include <boost/interprocess/windows_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  windows_shared_memory shdmem{open_or_create, "Boost", read_write, 1024};
  mapped_region region{shdmem, read_write};
  int *i1 = static_cast<int*>(region.get_address());
```

```
  *i1 = 99;
  mapped_region region2{shdmem, read_only};
  int *i2 = static_cast<int*>(region2.get_address());
  std::cout << *i2 << '\n';
}
```

Beachten Sie, dass die Klasse `boost::interprocess::windows_shared_memory` keine Methode `trunc ate()` anbietet. Stattdessen müssen Sie die Größe des Shared Memory als vierten Parameter dem Konstruktor übergeben.
Auch wenn die Klasse `boost::interprocess::windows_shared_memory` nicht portabel ist und nur unter Windows verwendet werden kann: Sie ist vor allem dann von Nutzen, wenn Sie mit anderen Windows-Anwendungen Daten austauschen wollen, die diese besondere Art von Shared Memory verwenden.

33.2 Verwalteter Shared Memory

Sie haben im vorherigen Abschnitt die Klasse `boost::interprocess::shared_memory_object` kennengelernt, mit der ein Shared Memory erstellt werden kann. In der Praxis arbeiten Sie eher selten mit dieser Klasse, weil Sie in diesem Fall selbst Bytes im Shared Memory schreiben und lesen müssen. Als C++-Entwickler sind Sie es gewohnt, Objekte vom Typ einer Klasse zu erstellen, ohne sich darum kümmern zu müssen, wie und wo die Bytes dieser Objekte im Speicher abgelegt sind.
Boost.Interprocess bietet ein Konzept an, das sich verwalteter Shared Memory nennt – auf Englisch managed shared memory. Es wird durch die Klasse `boost::interprocess::managed_shared_memory` zur Verfügung gestellt, die in der Headerdatei `boost/interprocess/managed_shared_memory.hpp` definiert ist. Mit Hilfe dieses verwalteten Shared Memory wird es möglich, Objekte derart zu instanziieren, dass sich der vom Objekt benötigte Speicher in einem Shared Memory befindet – und das Objekt für andere Prozesse, die auf den gleichen Shared Memory zugreifen, automatisch verfügbar wird.

Beispiel 33.6 Verwalteten Shared Memory verwenden

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  int *i = managed_shm.construct<int>("Integer")(99);
  std::cout << *i << '\n';
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
  if (p.first)
    std::cout << *p.first << '\n';
}
```

Im Beispiel 33.6 wird ein verwalteter Shared Memory namens Boost mit einer Größe von 1024 Bytes geöffnet. Wird kein Shared Memory mit diesem Namen gefunden, wird er automatisch erstellt.
Während Sie bei einem Shared Memory, wie Sie ihn im vorherigen Abschnitt kennengelernt haben, direkt auf einzelne Bytes zugreifen, um Daten zu speichern oder zu lesen, rufen Sie für einen verwalteten Shared Memory Methoden wie `construct()` auf. Diese Methode erwartet als Template-Parameter einen Typ – im Beispiel 33.6 ist int angegeben. Als Funktionsparameter wird ein Name übergeben, mit dem sich das entsprechende Objekt, das im verwalteten Shared Memory erstellt wird, finden lässt. So bekommt im Beispiel 33.6 die int-Variable den Namen Integer.
Da `construct()` ein Proxy-Objekt zurückgibt, können Sie über eine weitere Klammer Parameter an das Proxy-Objekt übergeben und es auf diese Weise initialisieren. Die Syntax ähnelt einem Konstruktoraufruf. Somit ist sichergestellt, dass Sie mit `construct()` nicht nur neue Objekte in einem verwalteten Shared Memory erstellen können, sondern sich diese auch wie gewünscht initialisieren lassen.
Um auf ein Objekt im verwalteten Shared Memory zuzugreifen, verwenden Sie die Methode `find()`. Sie übergeben Ihr den Namen des Objekts, das Sie suchen, und erhalten als Ergebnis einen Zeiger zurück. Wird kein Objekt mit dem angegebenen Namen gefunden, ist der Rückgabewert 0.

Sie sehen im Beispiel 33.6, dass `find()` den Zeiger in einem Objekt vom Typ `std::pair` zurückgibt. Der Zeiger wird nicht direkt, sondern über die Eigenschaft **`first`** zurückgegeben. Was Sie in **`second`** erhalten, lernen Sie anhand des folgenden Beispiels.

Beispiel 33.7 Arrays im verwalteten Shared Memory erstellen

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  int *i = managed_shm.construct<int>("Integer")[10](99);
  std::cout << *i << '\n';
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
  if (p.first)
  {
    std::cout << *p.first << '\n';
    std::cout << p.second << '\n';
  }
}
```

Im Beispiel 33.7 wird im Vergleich zum vorherigen nicht eine int-Variable erzeugt, sondern ein Array mit einer Größe von zehn Elementen. Dies geschieht, weil hinter dem Aufruf von `construct()` in eckigen Klammern die Zahl 10 angegeben ist. Beim Zugriff auf **`second`** wird genau diese `10` auf die Standardausgabe ausgegeben. Es ist daher möglich zu erkennen, ob es sich bei einem Objekt, das mit `find()` gefunden wurde, tatsächlich um ein einzelnes Objekt handelt oder um ein Array. Bei einzelnen Objekten ist **`second`** auf 1 gesetzt. Für Arrays ist die entsprechende Größe angegeben.
Beachten Sie, dass im Beispiel 33.7 alle zehn Stellen im Array mit der Zahl 99 initialisiert werden. Möchten Sie die Stellen eines Arrays mit unterschiedlichen Werten initialisieren, müssen Sie als Funktionsparameter einen Iterator übergeben.
Beachten Sie außerdem, dass `construct()` fehlschlägt, wenn es bereits ein Objekt mit dem angegebenen Namen im verwalteten Shared Memory gibt. In diesem Fall gibt `construct()` 0 zurück. Wollen Sie das Objekt für den Fall, dass es bereits existiert, wiederverwenden, rufen Sie `find_or_construct()` auf. Sollte das Objekt bereits existieren, wird nicht 0, sondern ein Zeiger auf das existierende Objekt zurückgegeben. In diesem Fall findet keine Initialisierung statt.
Die Methode `construct()` kann auch in einem anderen Fall fehlschlagen. Betrachten Sie dazu Beispiel 33.8.

Beispiel 33.8 `boost::interprocess::bad_alloc` im Fehlerfall

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  try
  {
    shared_memory_object::remove("Boost");
    managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
    int *i = managed_shm.construct<int>("Integer")[4096](99);
  }
  catch (boost::interprocess::bad_alloc &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Im Beispiel 33.8 wird versucht, ein Array vom Typ int zu erstellen, das 4096 Stellen umfasst. Der verwaltete Shared Memory ist jedoch lediglich 1024 Bytes groß. Der benötigte Speicher kann vom verwalteten Shared

Memory nicht zur Verfügung gestellt werden. Es wird eine Ausnahme vom Typ `boost::interprocess::bad_alloc` geworfen.
Nachdem Sie gesehen haben, wie Objekte in einem verwalteten Shared Memory erstellt und gefunden werden können, lernen Sie die Methode `destroy()` kennen, mit der Objekte gelöscht werden können.

Beispiel 33.9 Objekte aus dem verwalteten Shared Memory entfernen

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")(99);
  std::cout << *i << '\n';
  managed_shm.destroy<int>("Integer");
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
  std::cout << p.first << '\n';
}
```

Wie im Beispiel 33.9 zu sehen, übergeben Sie der Methode `destroy()` den Namen des Objekts, das Sie löschen möchten. Diese Methode gibt ein Ergebnis vom Typ bool zurück, falls Sie wissen möchten, ob ein Objekt mit dem angegebenen Namen gefunden und gelöscht wurde. Ein Objekt wird immer gelöscht, wenn es gefunden werden konnte. Gibt die Methode `false` zurück, bedeutet dies, dass kein Objekt mit dem angegebenen Namen gefunden werden konnte.
Neben `destroy()` steht eine Methode `destroy_ptr()` zur Verfügung, der Sie einen Zeiger auf ein Objekt im verwalteten Shared Memory übergeben können. Sie können `destroy_ptr()` auch für Arrays verwenden.
Nachdem Sie gesehen haben, wie einfach es mit verwaltetem Shared Memory ist, Objekte in einem mit anderen Prozessen gemeinsam genutzten Speicherbereich zu legen, möchten Sie womöglich auch Container aus der Standardbibliothek verwenden. Da Container dynamisch Speicher reservieren, muss ihnen mitgeteilt werden, dass dieser Speicher in einem verwalteten Shared Memory reserviert werden soll und nicht wie üblich mit `new` da, wo es das Betriebssystem für sinnvoll hält.
Viele Implementationen der Standardbibliothek sind nicht flexibel genug, um die von ihnen angebotenen Container wie `std::string` oder `std::list` mit Boost.Interprocess verwenden zu können. Das schließt die Implementationen der Standardbibliothek ein, die mit Visual C++ 2013, GCC und Clang ausgeliefert werden.
Um Entwicklern dennoch die Möglichkeit zu bieten, die aus dem Standard bekannten Container verwenden zu können, bietet Boost.Interprocess im Namensraum `boost::interprocess` flexible Implementationen der gleichen Klassen an. So steht zum Beispiel eine Klasse `boost::interprocess::string` zur Verfügung, die genauso funktioniert wie `std::string`. Der Vorteil ist, dass Objekte vom Typ `boost::interprocess::string` auf alle Fälle in einem verwalteten Shared Memory abgelegt werden können, selbst wenn das mit Objekten vom Typ `std::string` wie bei vielen heutigen Implementationen der Standardbibliothek nicht funktioniert.

Beispiel 33.10 Strings im verwalteten Shared Memory ablegen

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  typedef allocator<char,
    managed_shared_memory::segment_manager> CharAllocator;
  typedef basic_string<char, std::char_traits<char>, CharAllocator> string;
  string *s = managed_shm.find_or_construct<string>("String")("Hello!",
    managed_shm.get_segment_manager());
  s->insert(5, ", world");
```

```
  std::cout << *s << '\n';
}
```

Um wie im Beispiel 33.10 einen String im verwalteten Shared Memory erstellen zu können, der bei einem Aufruf von `insert()` den zusätzlich benötigten Speicher automatisch im gleichen verwalteten Shared Memory reserviert, muss ein entsprechender Typ definiert werden. Denn der Typ, auf dem der String basiert, darf nicht den Standardallokator aus dem Standard verwenden, sondern muss auf einen Allokator von Boost.Interprocess zugreifen.
Boost.Interprocess bietet eine Klasse `boost::interprocess::allocator` an, die in der Headerdatei `boost/interprocess/allocators/allocator.hpp` definiert ist. Mit dieser Klasse kann ein Allokator erstellt werden, der den *Segment-Manager* des verwalteten Shared Memory verwendet. Der Segment-Manager ist für die Verwaltung des Speichers in einem verwalteten Shared Memory verantwortlich. Mit dem neu definierten Allokator kann ein entsprechender Typ für den String definiert werden, wobei wie bereits erwähnt auf `boost::interprocess::basic_string` und nicht auf `std::basic_string` zugegriffen wird. Der neu definierte Typ – im Beispiel 33.10 `string` genannt – basiert auf der Implementation von `boost::interprocess::basic_string` und greift über den Allokator auf einen Segment-Manager zu. Damit die Instanz von `string`, die durch den Aufruf von `find_or_construct()` erstellt wird, weiß, auf welchen Segment-Manager sie zuzugreifen hat, wird der Zeiger auf den Segment-Manager als zweiter Parameter an den Konstruktor übergeben.
Neben `boost::interprocess::string` bietet Boost.Interprocess für viele andere aus der Standardbibliothek bekannte Container eigene Implementationen an. So stehen zum Beispiel Container wie `boost::interprocess::vector` und `boost::interprocess::map` zur Verfügung, die in den entsprechenden Headerdateien `boost/interprocess/containers/vector.hpp` und `boost/interprocess/containers/map.hpp` definiert sind.
Beachten Sie, dass die Container aus Boost.Container Boost.Interprocess unterstützen und in verwaltetem Shared Memory abgelegt werden können. Sie müssen nicht zwangsläufig auf die Container in `boost::interprocess` zugreifen. Boost.Container wird im Kapitel 20 vorgestellt.
Wenn Sie in mehreren Prozessen auf den gleichen verwalteten Shared Memory zugreifen und Objekte erstellen, suchen und zerstören, werden diese Operationen automatisch synchronisiert ausgeführt. Wenn zwei Prozesse zur gleichen Zeit versuchen, zwei Objekte mit unterschiedlichem Namen in einem verwalteten Shared Memory zu erstellen, wird der Zugriff auf den Speicher derart synchronisiert, dass beide Funktionen nacheinander ausgeführt werden. Möchten Sie mehrere Operationen gemeinsam ausführen, ohne dass diese eventuell von Operationen in einem gleichzeitig ausgeführten Prozess unterbrochen werden, bietet sich die Methode `atomic_func()` an. Sehen Sie sich dazu Beispiel 33.11 an.

Beispiel 33.11 Atomarer Zugriff auf verwalteten Shared Memory

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <functional>
#include <iostream>

using namespace boost::interprocess;

void construct_objects(managed_shared_memory &managed_shm)
{
  managed_shm.construct<int>("Integer")(99);
  managed_shm.construct<float>("Float")(3.14);
}

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  auto atomic_construct = std::bind(construct_objects,
    std::ref(managed_shm));
  managed_shm.atomic_func(atomic_construct);
  std::cout << *managed_shm.find<int>("Integer").first << '\n';
  std::cout << *managed_shm.find<float>("Float").first << '\n';
}
```

Sie übergeben der Methode `atomic_func()` als einzigen Parameter eine Funktion, die keinen Rückgabewert besitzt und keinen Parameter erwartet. Diese Funktion wird derart aufgerufen, dass Sie in ihr exklusiven Zugriff auf den verwalteten Shared Memory haben – jedenfalls, was die Methoden zum Erstellen, Suchen und Löschen

von Objekten betrifft. Hat zum Beispiel ein anderer Prozess bereits einen Zeiger auf ein Objekt im verwalteten Shared Memory erhalten, kann er über diesen Zeiger auf das Objekt zugreifen und es verändern.
Sie können mit Boost.Interprocess auch den Zugriff auf Objekte synchronisieren. Da Boost.Interprocess nicht wissen kann, wer wann auf die von Ihnen verwendeten Objekte wie zugreifen darf, müssen Sie die Zugriffe selbst synchronisieren. Die Klassen, die Boost.Interprocess zur Synchronisation zur Verfügung stellt, lernen Sie im nächsten Abschnitt kennen.

33.3 Synchronisation

Boost.Interprocess ermöglicht es mehreren Prozessen, einen Shared Memory gemeinsam zu nutzen. Da Prozesse gleichzeitig laufen, konkurrieren sie um den Zugriff auf Daten im Shared Memory. Weil ein Shared Memory eine gemeinsam genutzte Ressource ist, muss Boost.Interprocess die Synchronisation von Zugriffen unterstützen.
Wenn Sie an Synchronisation denken, denken Sie womöglich an Threads. So stehen sowohl in der C++11-Standardbibliothek als auch in Boost.Threads verschiedene Klassen zur Synchronisierung zur Verfügung. Diese Klassen können jedoch ausschließlich zur Synchronisierung von Threads im gleichen Prozess verwendet werden. Sie können sie nicht verwenden, um mehrere Prozesse zu synchronisieren. Da es sich jedoch grundsätzlich um das gleiche Problem handelt – in beiden Fällen geht es um einen synchronisierten Zugriff auf gemeinsam genutzte Ressourcen – werden Ihnen in diesem Abschnitt die gleichen Konzepte wiederbegegnen, die Sie von der C++11-Standardbibliothek oder von Boost.Thread kennen.
Während sich in Multithreaded-Anwendungen, die auf Boost.Thread zugreifen, Synchronisationsobjekte wie Mutexe und Bedingungsvariablen im gleichen Prozess befinden und somit allen Threads zur Verfügung stehen, gibt es bei Shared Memory das Problem, dass sich voneinander unabhängige Prozesse Synchronisationsobjekte teilen müssen. Wenn ein Prozess einen Mutex erstellt, müssen andere Prozesse auf diesen Mutex zugreifen können.
Boost.Interprocess bietet zwei Arten von Synchronisationsobjekten an: Anonyme Synchronisationsobjekte werden direkt im Shared Memory abgelegt, so dass sie dort automatisch allen Prozessen zugänglich sind. Anderen Synchronisationsobjekten werden Namen gegeben, so dass alle Prozesse über diese Namen auf die gleichen Synchronisationsobjekte zugreifen können. Diese Synchronisationsobjekte werden nicht im Shared Memory abgelegt, sondern vom Betriebssystem verwaltet.

Beispiel 33.12 Benannter Mutex mit `named_mutex`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")();
  named_mutex named_mtx{open_or_create, "mtx"};
  named_mtx.lock();
  ++(*i);
  std::cout << *i << '\n';
  named_mtx.unlock();
}
```

Im Beispiel 33.12 sehen Sie, wie Sie einen Mutex mit Namen erstellen und verwenden. Dazu wird auf die Klasse `boost::interprocess::named_mutex` zugegriffen, die in der Headerdatei `boost/interprocess/sync/named_mutex.hpp` definiert ist.
Dem Konstruktor der Klasse `boost::interprocess::named_mutex` müssen Sie neben einem Parameter, der angibt, ob der Mutex geöffnet oder erstellt werden soll, einen Namen übergeben. Jeder Prozess, der den Namen kennt, kann den gleichen Mutex öffnen. Um den Zugriff auf Daten im Shared Memory zu synchronisieren, muss ein Prozess lediglich den Mutex in Besitz nehmen, indem er `lock()` aufruft. Da ein Mutex immer nur von einem Prozess in Besitz genommen werden kann, muss ein anderer Prozess gegebenenfalls warten, bis der erste Prozess den Mutex mit `unlock()` freigegeben hat. Wenn ein Prozess einen Mutex in Besitz genommen hat, bedeutet das, dass er exklusiven Zugriff auf eine Ressource hat. Im Beispiel 33.12 ist diese Ressource eine int-Variable, die inkrementiert und auf die Standardausgabe ausgegeben wird.

Wenn Sie Beispiel 33.12 mehrfach starten, wird Ihnen bei jedem Programmstart ein um 1 erhöhter Wert ausgegeben. Auch dann, wenn Sie das Programm mehrfach gleichzeitig starten, ist dank dem Mutex sichergestellt, dass der Zugriff aller gleichzeitig laufender Prozesse auf den gleichen Shared Memory und die gleiche int-Variable synchronisiert stattfindet.

Beispiel 33.13 Anonymer Mutex mit `interprocess_mutex`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")();
  interprocess_mutex *mtx =
    managed_shm.find_or_construct<interprocess_mutex>("mtx")();
  mtx->lock();
  ++(*i);
  std::cout << *i << '\n';
  mtx->unlock();
}
```

Im Beispiel 33.13 wird ein anonymer Mutex vom Typ `boost::interprocess::interprocess_mutex` verwendet, der im Shared Memory abgelegt werden muss, um allen Prozessen zugänglich zu sein. Er ist in der Headerdatei `boost/interprocess/sync/interprocess_mutex.hpp` definiert.
Beispiel 33.13 funktioniert genauso wie das vorherige. Der einzige Unterschied ist, dass dieser Mutex im Shared Memory abgelegt ist. Dazu wird die Methode `construct()` oder `find_or_construct()` der Klasse `boost::interprocess::managed_shared_memory` verwendet.
Die Klassen `boost::interprocess::named_mutex` und `boost::interprocess::interprocess_mutex` bieten neben der Methode `lock()` zusätzlich `try_lock()` und `timed_lock()` an. Diese Methoden funktionieren genauso wie die gleichnamigen Methoden der Mutex-Klassen aus der C++11-Standardbibliothek und von Boost.Thread.
Für den Fall, dass Sie rekursive Mutexe einsetzen möchten, bietet Boost.Interprocess die beiden Klassen `boost::interprocess::named_recursive_mutex` und `boost::interprocess::interprocess_recursive_mutex` an.
Während Mutexe nützlich sind, um exklusiven Zugriff auf gemeinsam genutzte Ressourcen zu erhalten, helfen *Bedingungsvariablen* zu steuern, wer wann exklusiven Zugriff haben muss. Die von Boost.Interprocess zur Verfügung gestellten Bedingungsvariablen funktionieren grundsätzlich nicht anders als die Bedingungsvariablen in der C++11-Standardbibliothek und in Boost.Thread. Da die Klassen ähnliche Schnittstellen besitzen, sind sie entsprechend einfach zu verwenden.

Beispiel 33.14 Benannte Bedingungsvariable mit `named_condition`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <boost/interprocess/sync/named_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")(0);
  named_mutex named_mtx{open_or_create, "mtx"};
  named_condition named_cnd{open_or_create, "cnd"};
  scoped_lock<named_mutex> lock{named_mtx};
  while (*i < 10)
  {
    if (*i % 2 == 0)
    {
```

```
      ++(*i);
      named_cnd.notify_all();
      named_cnd.wait(lock);
    }
    else
    {
      std::cout << *i << std::endl;
      ++(*i);
      named_cnd.notify_all();
      named_cnd.wait(lock);
    }
  }
  named_cnd.notify_all();
  shared_memory_object::remove("shm");
  named_mutex::remove("mtx");
  named_condition::remove("cnd");
}
```

Im Beispiel 33.14 wird eine Bedingungsvariable vom Typ `boost::interprocess::named_condition` verwendet. Diese Klasse ist in der Headerdatei `boost/interprocess/sync/named_condition.hpp` definiert. Da es sich um eine Bedingungsvariable mit Namen handelt, muss sie nicht in einem Shared Memory abgelegt werden.
Das Programm verwendet eine `while`-Schleife, um eine im Shared Memory abgelegte Variable vom Typ int zu inkrementieren. Während die int-Variable in jedem Schleifendurchgang um 1 erhöht wird, wird sie lediglich in jedem zweiten Schleifendurchgang auf die Standardausgabe ausgegeben: Es werden lediglich ungerade Zahlen ausgegeben.
Jedes Mal, wenn die int-Variable um 1 erhöht wurde, wird auf die Bedingungsvariable **`named_cnd`** zugegriffen und `wait()` aufgerufen. Dieser Methode wird ein *Lock* übergeben – im Beispiel 33.14 ist dies die Variable **`lock`**. Der Lock basiert auf dem RAII-Idiom und nimmt einen Mutex im Konstruktor in Besitz. Im Destruktor wiederum wird der Mutex freigegeben. Der Lock wird vor Beginn der `while`-Schleife erstellt, womit der Mutex während der Ausführung des gesamten Programms in Besitz genommen ist. Wenn er jedoch als Parameter der Methode `wait()` übergeben wird, wird er automatisch freigegeben.
Bedingungsvariablen werden verwendet, um zu warten, bis jemand ein Signal schickt, dass die Warterei beendet werden kann. Diese Synchronisation erfolgt über die beiden Methoden `wait()` und `notify_all()`. Wenn ein Programm `wait()` aufruft, gibt es den entsprechende Mutex frei und wartet darauf, dass jemand `notify_all()` für die gleiche Bedingungsvariable aufruft.
Wenn Sie Beispiel 33.14 starten, stellen Sie fest, dass erst mal nicht viel zu passieren scheint: In der `while`-Schleife wird die int-Variable von 0 auf 1 erhöht. Anschließend wartet das Programm mit `wait()` auf ein Signal. Damit dieses Signal irgendwann erfolgt, starten Sie das gleiche Programm ein zweites Mal.
Die zweite Instanz des Programms wird versuchen, den gleichen Mutex in Besitz zu nehmen, bevor die `while`-Schleife gestartet wird. Das funktioniert, weil die erste Instanz den Mutex durch den Aufruf von `wait()` freigegeben hat. Die zweite Instanz kann entsprechend die `while`-Schleife ausführen. Da die erste Instanz die int-Variable im Shared Memory von 0 auf 1 gesetzt hat, wird der `else`-Zweig der `if`-Kontrollstruktur ausgeführt. Hier wird der aktuelle Wert der int-Variablen auf die Standardausgabe ausgeben, bevor sie anschließend um 1 erhöht wird.
Auch die zweite Instanz ruft `wait()` auf. Bevor dies geschieht – und das ist wichtig, damit die beiden Instanzen kooperieren – ruft sie `notify_all()` auf. Damit wird die erste Instanz informiert, dass der Aufruf von `wait()` zurückkehren kann. Die erste Instanz weiß, dass sie den Mutex wieder in Besitz nehmen kann, muss sich jedoch einen Augenblick gedulden, da der Mutex momentan von der zweiten Instanz in Besitz genommen ist. Da die zweite Instanz nach `notify_all()` als Nächstes `wait()` aufruft und damit den Mutex automatisch freigibt, kann die erste Instanz übernehmen.
So wechseln sich beide Instanzen ab und inkrementieren abwechselnd die int-Variable im Shared Memory. Lediglich eine Instanz aber gibt die Werte auf die Standardausgabe aus. Wenn die int-Variable die Zahl 10 speichert, wird die `while`-Schleife beendet. Damit die andere Instanz nicht endlos in `wait()` auf ein Signal wartet, wird nach der `while`-Schleife noch einmal `notify_all()` aufgerufen. Abschließend werden der verwendete Shared Memory, der Mutex und die Bedingungsvariable gelöscht.
So wie es zwei Arten von Mutexen gibt – einen anonymen, der im Shared Memory abgelegt werden muss, und einen, der über einen Namen identifiziert wird – gibt es auch zwei Arten von Bedingungsvariablen. Beispiel 33.14 wird nun derart umgeschrieben, dass eine anonyme Bedingungsvariable verwendet wird.

Beispiel 33.15 Anonyme Bedingungsvariable mit `interprocess_condition`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/interprocess_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")(0);
  interprocess_mutex *mtx =
    managed_shm.find_or_construct<interprocess_mutex>("mtx")();
  interprocess_condition *cnd =
    managed_shm.find_or_construct<interprocess_condition>("cnd")();
  scoped_lock<interprocess_mutex> lock{*mtx};
  while (*i < 10)
  {
    if (*i % 2 == 0)
    {
      ++(*i);
      cnd->notify_all();
      cnd->wait(lock);
    }
    else
    {
      std::cout << *i << std::endl;
      ++(*i);
      cnd->notify_all();
      cnd->wait(lock);
    }
  }
  cnd->notify_all();
  shared_memory_object::remove("shm");
}
```

Beispiel 33.15 funktioniert genauso wie das vorherige und muss ebenfalls zweimal gestartet werden, damit die int-Variable zehnmal inkrementiert wird.

Neben Mutexen und Bedingungsvariablen, die Sie in diesem Abschnitt kennengelernt haben, unterstützt Boost.Interprocess auch *Semaphore* und *File Locks*. Semaphore funktionieren ähnlich wie Bedingungsvariablen, nur dass sie nicht zwischen zwei Zuständen unterscheiden, sondern auf einem Zähler basieren. File Locks wiederum funktionieren wie Mutexe, wobei es sich bei ihnen nicht um Objekte im Speicher handelt, sondern um Dateien im Dateisystem.

So wie die C++11-Standardbibliothek und Boost.Thread verschiedene Typen von Mutexen und Locks unterscheiden, stehen auch in Boost.Interprocess mehrere Mutexe und Locks zur Verfügung. So gibt es neben den in den obigen Beispielen verwendeten Klassen Mutexe, die nicht nur exklusiv in Besitz genommen werden können. Dies ist nützlich, wenn mehrere Prozesse Daten gleichzeitig lesen wollen, da ein Mutex lediglich für Schreibvorgänge exklusiv in Besitz genommen werden muss. Entsprechend stehen verschiedene Klassen für Locks zur Verfügung, mit denen das RAII-Idiom auf die verschiedenen Mutexe angewandt werden kann.

Beachten Sie, dass Sie unbedingt unterschiedliche Namen verwenden sollten, wenn Sie nicht anonyme Synchronisationsobjekte verwenden. Obwohl Mutexe und Bedingungsvariablen auf unterschiedlichen Klassen basieren, gibt es in den von Boost.Interprocess abstrahierten Betriebssystemschnittstellen unter Umständen keine explizite Unterscheidung zwischen diesen Ressourcen. So werden zum Beispiel unter Windows für Mutexe und Bedingungsvariablen die gleichen Betriebssystemfunktionen verwendet. Wenn Sie Mutexen und Bedingungsvariablen den gleichen Namen geben, weil es sich scheinbar um unterschiedliche Objekte handelt, wird Ihr Programm unter Windows nicht wie erwartet funktionieren.

Teil VII

Streams und Dateien

Die folgenden Bibliotheken vereinfachen das Arbeiten mit Streams und Dateien.

- Boost.IOStreams stellt mit Streams und Filtern Konzepte und Implementationen zur Verfügung, die weit über das hinausgehen, was die Standardbibliothek rund um Streams anbietet. So haben Sie nicht nur Zugriff auf mehr Streams, um Daten aus unterschiedlichsten Quellen zu lesen und in unterschiedlichste Senken zu schreiben. Filter ermöglichen außerdem zum Beispiel, Daten über Streams komprimiert zu lesen oder zu schreiben.
- Boost.Filesystem bietet Zugriff aufs Dateisystem. So kann mit Boost.Filesystem zum Beispiel eine Datei kopiert oder über Dateien in einem Verzeichnis iteriert werden.

Kapitel 34

Boost.IOStreams

Dieses Kapitel stellt die Bibliothek Boost.IOStreams vor. Boost.IOStreams bricht die aus der Standardbibliothek bekannten Streams in mehrere Bestandteile auf. So definiert die Bibliothek ein Konzept namens *Device*, um ganz allgemein Datenquellen und -senken zu beschreiben. Neben Device kennt Boost.IOStreams ein Konzept namens *Stream*, das die von Streams aus der Standardbibliothek bekannte Schnittstelle zur formatierten Datenein- und -ausgabe zur Verfügung stellt. Ein Stream, so wie er von Boost.IOStreams definiert wird, ist jedoch nicht automatisch mit einer bestimmten Datenquelle oder -senke verbunden.
Boost.IOStreams bietet zahlreiche Implementationen der beiden Konzepte Device und Stream an. So existiert zum Beispiel mit `boost::iostreams::mapped_file` ein Device, um eine Datei ganz- oder teilweise in den Speicher zu laden. Mit `boost::iostreams::stream` wird eine Stream-Implementation angeboten, die mit einem Device wie `boost::iostreams::mapped_file` verknüpft wird, um über die bekannten Stream-Operatoren `operator<<` und `operator>>` Daten zu lesen oder zu schreiben.
Neben `boost::iostreams::stream` stellt die Bibliothek mit `boost::iostreams::filtering_stream` eine zweite Stream-Implementation zur Verfügung. Dieser Stream bietet die Möglichkeit, Daten zu filtern. So wird ein `boost::iostreams::filtering_stream` nicht nur mit einem Device verknüpft. Es können außerdem Filter hinzugefügt werden. Boost.IOStreams stellt zum Beispiel mit `boost::iostreams::gzip_compressor` einen Filter zur Verfügung, der Daten im GZIP-Format schreiben kann.
Boost.IOStreams kann auch eine Verbindung zu systemspezifischen Objekten herstellen. So existieren Devices, die mit einem Windows-Handle oder einem File Descriptor verknüpft werden können. Auf diese Weise können Objekte aus Low-Level-APIs in plattformunabhängigen C++-Code überführt werden.
Alle von Boost.IOStreams definierten Klassen und Funktionen befinden sich im Namensraum `boost::iostreams`. Eine Master-Headerdatei existiert nicht. Boost.IOStreams ist außerdem eine Bibliothek, die vorkompiliert wird, da sie nicht ausschließlich aus Headerdateien besteht. Das ist insofern wichtig, als dass abhängig von der Art und Weise, wie die Bibliothek vorkompiliert wurde, bestimmte Funktionen unter Umständen nicht unterstützt werden.

34.1 Devices

Devices sind Klassen, die Schreib- oder Lesezugriff auf Geräte bieten. Geräte sind üblicherweise externe Objekte wie Dateien. Es ist jedoch auch möglich, interne Objekte wie beispielsweise Arrays als Gerät zu verwenden.
Ein Device ist nicht mehr als eine Klasse, die eine Methode `read()` oder `write()` anbietet. Um nicht direkt mit diesen Methoden arbeiten zu müssen, sondern Daten formatiert ein- und ausgeben zu können, kann ein Device mit einem Stream verknüpft werden.

Beispiel 34.1 Ein Array mit `boost::iostreams::array_sink` als Device verwenden

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/stream.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
```

```
  stream<array_sink> os{sink};
  os << "Boost" << std::flush;
  std::cout.write(buffer, 5);
}
```

Im Beispiel 34.1 wird mit `boost::iostreams::array_sink` ein Device verwendet, um Daten in ein Array zu schreiben. Das Array wird als Parameter an den Konstruktor übergeben. Anschließend wird das Device mit einem Stream vom Typ `boost::iostreams::stream` verknüpft. Dazu ist es nicht nur notwendig, dem Konstruktor von `boost::iostreams::stream` eine Referenz auf das Device zu übergeben. Da es sich bei `boost::iostreams::stream` um ein Template handelt, muss außerdem der Typ des Devices als Parameter übergeben werden.
Im Beispiel wird über `operator<<` „Boost“ in den Stream geschrieben. Der Stream leitet die Daten an das Device weiter. Da das Device mit dem Array **`buffer`** verknüpft ist, wird „Boost“ in den ersten fünf Elementen im Array gespeichert. Zur Kontrolle werden diese mit `write()` auf die Standardausgabe ausgegeben. Führen Sie das Programm aus, wird `Boost` ausgegeben.

Beispiel 34.2 Ein Array mit `boost::iostreams::array_source` als Device verwenden

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/stream.hpp>
#include <string>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  stream<array_sink> os{sink};
  os << "Boost" << std::endl;

  array_source source{buffer};
  stream<array_source> is{source};
  std::string s;
  is >> s;
  std::cout << s << '\n';
}
```

Beispiel 34.2 baut auf dem vorherigen auf. So wird der String, der mit `boost::iostreams::array_sink` in ein Array geschrieben wird, mit `boost::iostreams::array_source` gelesen.
`boost::iostreams::array_source` wird genauso wie `boost::iostreams::array_sink` verwendet. Während `boost::iostreams::array_sink` ausschließlich Schreiboperationen unterstützt, kann mit `boost::iostreams::array_source` nur gelesen werden. Mit `boost::iostreams::array` bietet Boost.IOStreams auch ein Device an, das Schreib- und Leseoperationen unterstützt.
Beachten Sie, dass `boost::iostreams::array_source` und `boost::iostreams::array_sink` eine Referenz auf ein Array erhalten. Das Array darf nicht zerstört werden, solange die Devices verwendet werden sollen.

Beispiel 34.3 Einen Vektor mit `back_insert_device` als Device verwenden

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/device/back_inserter.hpp>
#include <boost/iostreams/stream.hpp>
#include <vector>
#include <string>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  std::vector<char> v;
  back_insert_device<std::vector<char>> sink{v};
  stream<back_insert_device<std::vector<char>>> os{sink};
  os << "Boost" << std::endl;
```

```
  array_source source{v.data(), v.size()};
  stream<array_source> is{source};
  std::string s;
  is >> s;
  std::cout << s << '\n';
}
```

Beispiel 34.3 verwendet anstatt `boost::iostreams::array_sink` ein Device vom Typ `boost::iostreams::back_insert_device`. Dieses Device kann verwendet werden, um Daten in beliebige Container zu schreiben. Dies geschieht über die Methode `insert()`, die vom Device aufgerufen wird und vom Container angeboten werden muss.
Im Beispiel wird `boost::iostreams::back_insert_device` verwendet, um „Boost“ in einen Vektor zu schreiben. „Boost“ wird anschließend mit Hilfe von `boost::iostreams::array_source` ausgelesen. Dazu werden ein Zeiger auf den Anfang des Vektors und die Größe des Vektors als Parameter an den Konstruktor übergeben. Wenn Sie das Beispielprogramm ausführen, wird wie zuvor `Boost` ausgegeben.

Beispiel 34.4 Dateien mit `boost::iostreams::file_source` als Device verwenden

```
#include <boost/iostreams/device/file.hpp>
#include <boost/iostreams/stream.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  file_source f{"main.cpp"};
  if (f.is_open())
  {
    stream<file_source> is{f};
    std::cout << is.rdbuf() << '\n';
    f.close();
  }
}
```

Im Beispiel 34.4 wird das Device `boost::iostreams::file_source` verwendet, mit dem Dateien gelesen werden können. Während die bisher vorgestellten Devices neben dem Konstruktor keine weiteren Methoden anbieten, kann für `boost::iostreams::file_source` `is_open()` aufgerufen werden, um zu überprüfen, ob die gewünschte Datei geöffnet werden konnte. Mit `close()` steht außerdem eine Methode zur Verfügung, um das Device explizit zu schließen. Ein Aufruf von `close()` ist nicht zwingend notwendig, da `boost::iostreams::file_source` automatisch im Destruktor geschlossen wird.
Neben `boost::iostreams::file_source` bietet Boost.IOStreams mit `boost::iostreams::mapped_file_source` ein Device an, um eine Datei ganz- oder teilweise in den Speicher zu laden. `boost::iostreams::mapped_file_source` stellt hierzu eine Methode `data()` zur Verfügung, um einen Zeiger auf den entsprechenden Speicherbereich zu erhalten. So kann direkt auf beliebige Daten in einer Datei zugegriffen werden, ohne sie sequentiell lesen zu müssen.
Neben `boost::iostreams::file_source` und `boost::iostreams::mapped_file_source` bietet Boost.IOStreams die Devices `boost::iostreams::file_sink` und `boost::iostreams::mapped_file_sink` an, um schreibend auf Dateien zuzugreifen.

Beispiel 34.5 `file_descriptor_source` und `file_descriptor_sink` in Aktion

```
#include <boost/iostreams/device/file_descriptor.hpp>
#include <boost/iostreams/stream.hpp>
#include <iostream>
#include <Windows.h>

using namespace boost::iostreams;

int main()
{
  HANDLE handles[2];
  if (CreatePipe(&handles[0], &handles[1], nullptr, 0))
```

```
  {
    file_descriptor_source src{handles[0], close_handle};
    stream<file_descriptor_source> is{src};

    file_descriptor_sink snk{handles[1], close_handle};
    stream<file_descriptor_sink> os{snk};

    os << "Boost" << std::endl;
    std::string s;
    std::getline(is, s);
    std::cout << s;
  }
}
```

Im Beispiel 34.5 kommen die Devices `boost::iostreams::file_descriptor_source` und `boost::iostreams::file_descriptor_sink` zum Einsatz. Diese Devices ermöglichen Lese- und Schreiboperationen auf systemspezifische Objekte. Unter Windows sind dies Handles, unter POSIX-Betriebssystemen File Descriptoren. Im Beispiel wird auf die Windows-Funktion `CreatePipe()` zugegriffen, um eine Pipe zu erstellen. Die Lese- und Schreib-Enden der Pipe werden im Array **`handles`** erhalten. Das Lese-Ende wird einem Device vom Typ `boost::iostreams::file_descriptor_source`, das Schreib-Ende einem Device vom Typ `boost::iostreams::file_descriptor_sink` übergeben. Alles, was daraufhin in den Stream **`os`** geschrieben wird, der mit dem Schreib-Ende verknüpft ist, kann über den Stream **`is`**, der mit dem Lese-Ende verknüpft ist, gelesen werden. So wird im Beispiel 34.5 „Boost“ durch die Pipe geschickt und auf die Standardausgabe ausgegeben.
Beachten Sie, dass die Konstruktoren von `boost::iostreams::file_descriptor_source` und `boost::iostreams::file_descriptor_sink` zwei Parameter erwarten. Der erste Parameter ist ein Windows-Handle oder – wenn ein Programm für POSIX-Betriebssysteme erstellt wird – ein File Descriptor. Als zweiter Parameter muss **`boost::iostreams::close_handle`** oder **`boost::iostreams::never_close_handle`** übergeben werden. Dieser Parameter gibt an, ob der Destruktor der Devices den Windows-Handle oder File Descriptor automatisch schließen soll.

34.2 Filter

Neben Devices bietet Boost.IOStreams Filter an. Sie werden vor Devices gesetzt, um Daten, die von Devices gelesen oder in Devices geschrieben werden, zu filtern. In den folgenden Beispielprogrammen wird dazu auf die Klassen `boost::iostreams::filtering_istream` und `boost::iostreams::filtering_ostream` zugegriffen. Die Klassen ersetzen `boost::iostreams::stream`, da diese Klasse keine Filter unterstützt.

Beispiel 34.6 Daten mit `boost::iostreams::regex_filter` filtern

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/regex.hpp>
#include <boost/regex.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  filtering_ostream os;
  os.push(regex_filter{boost::regex{"Bo+st"}, "C++"});
  os.push(sink);
  os << "Boost" << std::flush;
  os.pop();
  std::cout.write(buffer, 3);
}
```

Im Beispiel 34.6 wird das Device `boost::iostreams::array_sink` verwendet, um Daten in ein Array zu schreiben. Die Daten sollen durch einen Filter vom Typ `boost::iostreams::regex_filter` geschickt wer-

den, der Zeichen ersetzen kann. Dieser Filter erwartet einen regulären Ausdruck und eine Formatierung. Der reguläre Ausdruck beschreibt, was ersetzt werden soll. Die Formatierung gibt an, durch was die gefundenen Zeichen ersetzt werden sollen. Im Beispiel soll „Boost“ durch „C++“ ersetzt werden. Dabei kann „Boost“ mit einem oder mehreren „o“ geschrieben werden.
Der Filter und das Device werden mit Hilfe des Streams `boost::iostreams::filtering_ostream` verknüpft. Diese Klasse bietet eine Methode `push()` an, der der Filter und das Device übergeben werden.
Beachten Sie, dass zuerst der Filter und dann das Device übergeben wird. Die Reihenfolge ist entscheidend. Sie können ein oder mehrere Filter mit `push()` übergeben. Wird ein Device übergeben, ist der Stream vollständig, und Sie dürfen `push()` kein weiteres Mal aufrufen.
Der Filter `boost::iostreams::regex_filter` kann Daten nicht schrittweise verarbeiten. Er kann Zeichen nicht einzeln filtern, da er auf einem regulären Ausdruck basiert und Zeichengruppen betrachten muss. Daher wird `boost::iostreams::regex_filter` erst aktiv, wenn ein Schreibvorgang abgeschlossen ist und alle Daten vorliegen. Dies ist der Fall, wenn ein Device mit `pop()` vom Stream entfernt wird. Im Beispiel 34.6 wird diese Methode aufgerufen, nachdem „Boost“ in den Stream geschrieben wurde. Ohne den Aufruf von `pop()` würde `boost::iostreams::regex_filter` keine Daten verarbeiten und daher auch keine Daten an das Device weiterleiten.
Beachten Sie, dass Sie einen Stream nicht verwenden dürfen, wenn er nicht mit einem Device verknüpft ist. Sie können den Stream jedoch jederzeit wieder vervollständigen, wenn Sie nach dem Aufruf von `pop()` mit `push()` ein Device hinzufügen.
Wenn Sie Beispiel 34.6 ausführen, wird `C++` ausgegeben.

Beispiel 34.7 Auf Filter in `boost::iostreams::filtering_ostream` zugreifen

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/counter.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  filtering_ostream os;
  os.push(counter{});
  os.push(sink);
  os << "Boost" << std::flush;
  os.pop();
  counter *c = os.component<counter>(0);
  std::cout << c->characters() << '\n';
  std::cout << c->lines() << '\n';
}
```

Beispiel 34.7 verwendet den Filter `boost::iostreams::counter`, der Zeichen und Zeilen zählt. Die Klasse bietet entsprechend die beiden Methoden `characters()` und `lines()` an.
Um auf einen Filter zugreifen zu können, stellt `boost::iostreams::filtering_stream` die Methode `component()` zur Verfügung. Ihr muss der Index des entsprechenden Filters als Parameter übergeben werden. Da es sich bei `component()` außerdem um ein Template handelt, muss der Typ des entsprechenden Filters als Template-Parameter angegeben werden. `component()` gibt einen Zeiger auf den entsprechenden Filter zurück. Der Zeiger ist 0, wenn ein falscher Filtertyp als Template-Parameter angegeben wird.
Im Beispiel 34.7 werden fünf Zeichen in den Stream geschrieben. Keines davon ist ein „\n“. Daher gibt das Programm `5` und `0` aus.

Beispiel 34.8 Mit ZLIB komprimierte Daten schreiben und lesen

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/device/back_inserter.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/zlib.hpp>
#include <vector>
#include <string>
#include <iostream>
```

```
using namespace boost::iostreams;

int main()
{
  std::vector<char> v;
  back_insert_device<std::vector<char>> snk{v};
  filtering_ostream os;
  os.push(zlib_compressor{});
  os.push(snk);
  os << "Boost" << std::flush;
  os.pop();

  array_source src{v.data(), v.size()};
  filtering_istream is;
  is.push(zlib_decompressor{});
  is.push(src);
  std::string s;
  is >> s;
  std::cout << s << '\n';
}
```

Im Beispiel 34.8 kommt neben `boost::iostreams::filtering_ostream` der Stream `boost::iostreams::filtering_istream` zum Einsatz. Dieser wird verwendet, um Daten mit Filtern zu lesen. So sollen in diesem Beispiel Daten komprimiert geschrieben und wieder gelesen werden.
Boost.IOStreams bietet zahlreiche Filter zur Datenkompression an. Die Klasse `boost::iostreams::zlib_compressor` kann verwendet werden, um Daten im ZLIB-Format zu komprimieren. Um im ZLIB-Format komprimierte Daten zu entpacken, wird die Klasse `boost::iostreams::zlib_decompressor` verwendet. Die Filter werden entsprechend mit `push()` den Streams hinzugefügt.
Wenn Sie Beispiel 34.8 ausführen, wird „Boost“ komprimiert in den Vektor **v** geschrieben und dekomprimiert in den String **s** gelesen. Das Programm gibt daher `Boost` auf die Standardausgabe aus.

> **Anmerkung**
>
> Beachten Sie, dass Boost.IOStreams unter Windows die Komprimierung mit ZLIB standardmäßig nicht unterstützt. Bei Boost.IOStreams handelt es sich um eine vorkompilierte Bibliothek, die unter Windows mit dem Makro `NO_ZLIB` erstellt wird. Sie müssen Makros wie `ZLIB_LIBPATH` und `ZLIB_SOURCE` setzen, um ZLIB-Unterstützung unter Windows zu erhalten.

Kapitel 35

Boost.Filesystem

Boost.Filesystem vereinfacht das Arbeiten mit Dateien und Verzeichnissen. Die Bibliothek stellt eine Klasse `boost::filesystem::path` zur Verfügung, mit der Pfadangaben verarbeitet werden können. Darüber hinaus existieren zahlreiche freistehende Funktionen, um zum Beispiel Verzeichnisse zu erstellen oder zu überprüfen, ob eine Datei existiert.
Boost.Filesystem wurde mehrfach überarbeitet. Im Folgenden wird die aktuelle Version Boost.Filesystem 3 vorgestellt, die seit Boost 1.46.0 die Standardversion ist. Boost.Filesystem 2 wurde letztmalig mit der Version 1.49.0 ausgeliefert.

35.1 Pfadangaben

Die Klasse `boost::filesystem::path` ist die zentrale Klasse in Boost.Filesystem. Sie repräsentiert Pfadangaben und bietet zahlreiche Methoden an, um Pfadangaben zu verarbeiten.
Die Headerdatei, die Sie zur Verarbeitung von Pfadangaben einbinden müssen, ist `boost/filesystem.hpp`. Alle von Boost.Filesystem angebotenen Klassen und Funktionen befinden sich im Namensraum `boost::filesystem`.
Sie können Pfadangaben bilden, indem Sie die Klasse `boost::filesystem::path` wie im Beispiel 35.1 mit einem entsprechenden String initialisieren.

Beispiel 35.1 `boost::filesystem::path` in Aktion

```
#include <boost/filesystem.hpp>

using namespace boost::filesystem;

int main()
{
  path p1{"C:\\"};
  path p2{"C:\\Windows"};
  path p3{L"C:\\Boost C++ \u5E93"};
}
```

Sie können `boost::filesystem::path` mit Wide-Strings initialisieren. Boost.Filesystem interpretiert Wide-Strings im Unicode-Format, so dass Pfadangaben erstellt werden können, die exotische Zeichen enthalten. Dies ist ein entscheidender Unterschied zum Vorgänger Boost.Filesystem 2, der mehrere Klassen wie `boost::filesystem::path` und `boost::filesystem::wpath` für unterschiedliche String-Typen anbot.
Beachten Sie, dass Boost.Filesystem keine Pfadangaben vom Typ `std::u16string` oder `std::u32string` unterstützt. Ihr Compiler bricht mit einer Fehlermeldung ab, wenn Sie versuchen, eine Pfadangabe von einem dieser String-Typen an den Konstruktor von `boost::filesystem::path` zu übergeben.
Die Konstruktoren von `boost::filesystem::path` überprüfen nicht, ob eine Pfadangabe Sinn ergibt und ob eine entsprechende Datei oder ein Verzeichnis existiert. Selbst mit offensichtlich unsinnigen Pfadangaben kann `boost::filesystem::path` instanziiert werden.

Beispiel 35.2 Unsinnige Pfadangaben mit `boost::filesystem::path`

```
#include <boost/filesystem.hpp>
```

```
using namespace boost::filesystem;

int main()
{
  path p1{"..."};
  path p2{"\\"};
  path p3{"@:"};
}
```

Beispiel 35.2 kann ohne Probleme ausgeführt werden, weil Pfadangaben letztendlich nur Strings sind. Die Klasse `boost::filesystem::path` macht nichts anderes als Strings zu verarbeiten. Es finden keine Zugriffe aufs Dateisystem statt.
Da `boost::filesystem::path` Strings verarbeitet, stehen Methoden bereit, um eine Pfadangabe als String zu erhalten. Interessanterweise stehen dazu eine ganze Reihe von Methoden zur Verfügung.
Boost.Filesystem unterscheidet grundsätzlich zwischen nativen und generischen Pfadangaben. *Native Pfadangaben* sind betriebssystemspezifisch und müssen verwendet werden, wenn zum Beispiel Betriebssystemfunktionen aufgerufen werden. *Generische Pfadangaben* sind portabel und betriebssystemunabhängig.

Beispiel 35.3 Pfadangaben von `boost::filesystem::path` als Strings erhalten

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\System"};

#ifdef BOOST_WINDOWS_API
  std::wcout << p.native() << '\n';
#else
  std::cout << p.native() << '\n';
#endif

  std::cout << p.string() << '\n';
  std::wcout << p.wstring() << '\n';

  std::cout << p.generic_string() << '\n';
  std::wcout << p.generic_wstring() << '\n';
}
```

Die ersten drei aufgerufenen Methoden `native()`, `string()` und `wstring()` geben die Pfadangabe im nativen Format zurück. Wenn Sie Beispiel 35.3 unter Windows ausführen, wird dreimal `C:\Windows\System` ausgegeben – eine Pfadangabe, wie Sie unter Windows üblich ist.
Die letzten beiden aufgerufenen Methoden `generic_string()` und `generic_wstring()` geben die Pfadangabe im generischen Format zurück. Dabei handelt es sich um eine portable Pfadangabe – der String wird normalisiert. Die Regeln, nach denen generische Pfadangaben gebildet werden, entsprechen denen des POSIX-Standards. Generische Pfadangaben sind daher identisch mit Pfadangaben, wie sie beispielsweise unter Linux verwendet werden. So wird zum Beispiel als Trennzeichen für Verzeichnisse der Schrägstrich verwendet. Beispiel 35.3 unter Windows ausgeführt gibt für `generic_string()` und `generic_wstring()` `C:/Windows/System` aus.
Der Rückgabewert von Methoden für native Pfadangaben hängt davon ab, unter welchem Betriebssystem ein Programm ausgeführt wird. Der Rückgabewert von Methoden für generische Pfadangaben ist unabhängig vom Betriebssystem. Somit helfen generische Pfadangaben, plattformunabhängigen Code zu schreiben, da sie unabhängig vom Betriebssystem Dateien und Verzeichnisse eindeutig identifizieren.
Da `boost::filesystem::path` mit unterschiedlichen String-Typen initialisiert werden kann, werden entsprechend mehrere Methoden angeboten, um Pfadangaben in unterschiedlichen String-Typen zu erhalten. Während `string()` und `generic_string()` einen String vom Typ `std::string` zurückgeben, geben `wstring()` und `generic_wstring()` einen String vom Typ `std::wstring` zurück.
Der Typ des Rückgabewerts von `native()` hängt vom Betriebssystem ab, für das das Programm kompiliert wird. So gibt `native()` unter Windows einen String vom Typ `std::wstring` und unter Linux einen String

vom Typ `std::string` zurück.
Beachten Sie, dass der Konstruktor von `boost::filesystem::path` sowohl generische als auch plattformabhängige Pfadangaben unterstützt. Die Angabe „C:\\Windows\\System“, die im obigen Programm verwendet wird, ist nicht portabel, sondern windows-spezifisch. Sie wird von Boost.Filesystem richtig interpretiert, wenn das Programm unter Windows ausgeführt wird. Das gleiche Programm auf einem POSIX-System wie Linux ausgeführt gibt für alle Methodenaufrufe `C:\Windows\System` zurück. Denn in diesem Fall erkennt Boost.Filesystem nicht, dass der Backslash als Trennzeichen für Dateien und Verzeichnisse verwendet werden soll: Er ist weder im portablen noch im nativen Format unter Linux ein Trennzeichen.
Das Makro `BOOST_WINDOWS_API` stammt aus Boost.System und ist definiert, wenn das Beispiel unter Windows kompiliert wird. Das entsprechende Makro für POSIX-Betriebssysteme lautet `BOOST_POSIX_API`.
Im Beispiel 35.4 findet die Initialisierung mit einem portablen Pfad statt.

Beispiel 35.4 `boost::filesystem::path` mit portablem Pfad initialisieren

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"/"};
  std::cout << p.string() << '\n';
  std::cout << p.generic_string() << '\n';
}
```

Da `generic_string()` einen portablen Pfad zurückgibt, ist das in diesem Fall genau die Angabe, mit der `boost::filesystem::path` initialisiert wurde – also „/“. Die Methode `string()` kann jedoch je nach Plattform unterschiedliche Werte zurückgeben. Unter Windows und Linux gibt diese Methode als Ergebnis „/“ zurück. Das Ergebnis ist für diese beiden Plattformen identisch, weil auch Windows den Schrägstrich als Trennzeichen für Verzeichnisse akzeptiert, auch wenn der Backslash das bevorzugte Trennzeichen ist.
`boost::filesystem::path` bietet mehrere Methoden an, um auf bestimmte Teile eines Pfads zuzugreifen.

Beispiel 35.5 Zugriff auf Teilausdrücke in Pfadangaben

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\System"};
  std::cout << p.root_name() << '\n';
  std::cout << p.root_directory() << '\n';
  std::cout << p.root_path() << '\n';
  std::cout << p.relative_path() << '\n';
  std::cout << p.parent_path() << '\n';
  std::cout << p.filename() << '\n';
}
```

Wenn Beispiel 35.5 unter Windows ausgeführt wird, wird der String „C:\\Windows\\System“ als plattformabhängige Pfadangabe interpretiert. Daraufhin gibt `root_name()` `"C:"` zurück, `root_directory()` `"\"`, `root_path()` `"C:\"`, `relative_path()` `"Windows\System"`, `parent_path()` `"C:\Windows"` und `filename()` `"System"`.
Die im Beispiel aufgerufenen Methoden geben plattformabhängige Pfadangaben zurück. `boost::filesystem::path` speichert Pfadangaben intern im plattformabhängigen Format. Möchten Sie Pfadangaben im portablen Format erhalten, müssen Sie explizit Methoden wie `generic_string()` aufrufen.
Wenn Sie Beispiel 35.5 unter Linux ausführen, erhalten Sie andere Werte. So geben fast alle Methoden eine leere Zeichenkette zurück. Einzige Ausnahme sind `relative_path()` und `filename()`, die jeweils `"C:\Windows\System"` zurückgeben. Die Angabe „C:\\Windows\\System“ wird demnach unter Linux als ein Dateiname interpretiert. Das ist insofern verständlich als dass es sich hierbei weder um eine portable Kodierung einer Pfadangabe noch um eine von Linux unterstützte plattformabhängige Kodierung handelt. Boost.Filesystem bleibt unter Linux nichts anderes übrig als die gesamte Angabe als Dateinamen zu interpretieren.

Boost.Filesystem bietet zusätzlich Methoden an, mit denen sich ermitteln lässt, ob das betreffende Teilstück in einem Pfad existiert. Diese Methoden lauten `has_root_name()`, `has_root_directory()`, `has_root_path()`, `has_relative_path()`, `has_parent_path()` und `has_filename()`. Sie alle haben einen Rückgabewert vom Typ bool.
Es gibt zwei weitere Methoden, mit denen ein Dateiname in seine Bestandteile zerlegt wird. Diese Methoden sollten nur dann aufgerufen werden, wenn `has_filename()` `true` zurückgibt. Ansonsten werden leere Strings zurückgegeben – wo kein Dateiname, kann nichts zerlegt werden.

Beispiel 35.6 Dateinamen und -endung erhalten

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"photo.jpg"};
  std::cout << p.stem() << '\n';
  std::cout << p.extension() << '\n';
}
```

Beispiel 35.6 gibt für `stem()` `"photo"` und für `extension()` `".jpg"` zurück.
Es ist auch möglich, über Bestandteile eines Pfads zu iterieren.

Beispiel 35.7 Über Bestandteile eines Pfads iterieren

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\System"};
  for (const path &pp : p)
    std::cout << pp << '\n';
}
```

Beispiel 35.7 gibt nacheinander `"C:"`, `"/"`, `"Windows"` und `"System"` aus – wenn das Programm unter Windows ausgeführt wird. Auf einem anderen Betriebssystem wie Linux lautet die Ausgabe `"C:\Windows\System"`.
Nachdem Sie mehrere Methoden kennengelernt haben, um auf verschiedene Bestandteile eines Pfads zuzugreifen, lernen Sie im Beispiel 35.8 eine Methode kennen, mit der eine Pfadangabe geändert werden kann.

Beispiel 35.8 Pfadangaben mit `operator/=` aneinanderhängen

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\"};
  p /= "Windows\\System";
  std::cout << p.string() << '\n';
}
```

Im Beispiel 35.8 wird mit Hilfe des überladenen Operators `operator/=` ein Pfad an einen anderen gehängt. Das Beispiel gibt daher `C:\Windows\System` aus – unter Windows. Unter Linux wird `C:\/Windows\System` ausgegeben, da unter Linux der Schrägstrich das Trennzeichen für Dateien und Verzeichnisse ist. Dieser Schrägstrich ist auch der Grund, warum der Operator `operator/=` überladen wurde.
Neben `operator/=` bietet Boost.Filesystem zum Ändern einer Pfadangabe lediglich die Methoden `remove_filename()`, `replace_extension()` und `make_preferred()` an. Letztgenannte Methode ist speziell für den Einsatz unter Windows gedacht.

Beispiel 35.9 Bevorzugte Pfadangaben mit `make_preferred()`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:/Windows/System"};
  std::cout << p.make_preferred() << '\n';
}
```

Auch wenn Pfadangaben unter Windows normalerweise den Backslash als Trennzeichen verwenden, akzeptiert Windows den Schrägstrich. So ist „C:/Windows/System“ eine gültige native Pfadangabe unter Windows. Mit `make_preferred()` kann die Pfadangabe in die bevorzugte Schreibweise umgewandelt werden. So gibt Beispiel 35.9 `"C:\Windows\System"` aus.
Unter Linux und anderen POSIX-Plattformen ändert `make_preferred()` eine Pfadangabe nicht.
Beachten Sie, dass `make_preferred()` nicht nur die geänderte Pfadangabe zurückgibt, sondern auch das Objekt ändert, für das die Methode aufgerufen wird. Die nach dem Methodenaufruf intern in **p** gespeicherte Pfadangabe ist „C:\Windows\System“.

35.2 Dateien und Verzeichnisse

Die Methoden, die Ihnen im Zusammenhang mit der Klasse `boost::filesystem::path` vorgestellt wurden, tun nichts anderes als Strings zu verarbeiten. So können Sie auf Teilbereiche eines Pfads zugreifen oder einen Pfad an einen anderen hängen. Intern werden in der Klasse `boost::filesystem::path` einfach nur Strings verarbeitet.
Um tatsächlich auf Dateien und Verzeichnisse zugreifen zu können, stehen zahlreiche freistehende Funktionen zur Verfügung. Diese erwarten als Parameter ein oder mehrere Objekte vom Typ `boost::filesystem::path` und rufen für die derart identifizierten Dateien oder Verzeichnisse Betriebssystemfunktionen auf.
Bevor verschiedene Funktionen zum Datei- und Verzeichniszugriff vorgestellt werden, erfahren Sie, was im Falle eines Fehlers passiert. Denn alle Funktionen, die Sie kennenlernen werden, greifen auf Betriebssystemfunktionen zu, die fehlschlagen können. Boost.Filesystem bietet von allen Funktionen zwei Varianten an, deren Verhalten sich im Fehlerfall unterscheidet:

- Die erste Variante wirft im Fehlerfall eine Ausnahme vom Typ `boost::filesystem::filesystem_error`. Diese Klasse ist von `boost::system::system_error` abgeleitet und fügt sich ins Framework von Boost.System ein.
- Die zweite Variante erwartet als zusätzlichen Parameter ein Objekt vom Typ `boost::system::error_code`. Dieses Objekt wird als Referenz übergeben und kann nach dem Funktionsaufruf ausgewertet werden. Im Fehlerfall enthält dieses Objekt einen Fehlercode.

`boost::system::system_error` und `boost::system::error_code` werden im Kapitel 55 vorgestellt. `boost::filesystem::filesystem_error` bietet zusätzlich zu der von `boost::system::system_error` geerbten Schnittstelle die Methoden `path1()` und `path2()` an. Beide geben ein Objekt vom Typ `boost::filesystem::path` zurück. Da es Funktionen gibt, die als Parameter zwei Objekte vom Typ `boost::filesystem::path` erwarten, stehen entsprechend zwei Methoden zur Verfügung. So können im Fehlerfall die Pfadangaben einfach ermittelt werden.
Im Beispiel 35.10 lernen Sie eine Funktion kennen, mit der der Status einer Datei oder eines Verzeichnisses abgefragt werden kann.

Beispiel 35.10 Dateistatus mit `boost::filesystem::status()` abfragen

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
```

```
{
  path p{"C:\\"};
  try
  {
    file_status s = status(p);
    std::cout << std::boolalpha << is_directory(s) << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

`boost::filesystem::status()` gibt ein Objekt vom Typ `boost::filesystem::file_status` zurück. Sie können dieses Objekt an verschiedene Hilfsfunktionen übergeben, um es auszuwerten. So gibt `boost::filesystem::is_directory()` `true` zurück, wenn der Status für ein Verzeichnis ermittelt wurde. Neben `boost::filesystem::is_directory()` stehen weitere Funktionen wie `boost::filesystem::is_regular_file()`, `boost::filesystem::is_symlink()` und `boost::filesystem::exists()` zur Verfügung, die alle einen Rückgabewert vom Typ bool besitzen.
Neben `boost::filesystem::status()` existiert eine zweite Funktion `boost::filesystem::symlink_status()`. Diese kann eingesetzt werden, wenn der Status einer Datei ermittelt werden soll, die eine symbolische Verknüpfung ist. Wenn für eine derartige Datei `boost::filesystem::status()` aufgerufen wird, wird der Status für die Datei ermittelt, auf die die symbolische Verknüpfung verweist. Unter Windows sind Dateien, die symbolische Verknüpfungen darstellen, an der Endung `lnk` zu erkennen.
Neben den eben vorgestellten Status-Funktionen gibt es eine andere Kategorie von Funktionen, mit denen Attribute ermittelt werden können.

Beispiel 35.11 Dateigröße mit `boost::filesystem::file_size()` ermitteln

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\win.ini"};
  boost::system::error_code ec;
  boost::uintmax_t filesize = file_size(p, ec);
  if (!ec)
    std::cout << filesize << '\n';
  else
    std::cout << ec << '\n';
}
```

Die Funktion `boost::filesystem::file_size()` gibt die Größe einer Datei in Bytes zurück. Der Rückgabewert hat den Typ boost::uintmax_t, der aus der Bibliothek Boost.Integer stammt.
Im Beispiel 35.11 wird außerdem ein Objekt vom Typ `boost::system::error_code` verwendet, das explizit ausgewertet werden muss, um zu überprüfen, ob der Aufruf von `boost::filesystem::file_size()` fehlschlug.

Beispiel 35.12 Zugriffszeit mit `boost::filesystem::last_write_time()` ermitteln

```
#include <boost/filesystem.hpp>
#include <iostream>
#include <ctime>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\win.ini"};
  try
  {
    std::time_t t = last_write_time(p);
```

```
    std::cout << std::ctime(&t) << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Um den Zeitpunkt zu ermitteln, wann eine Datei das letzte Mal geändert wurde, kann die im Beispiel 35.12 verwendete Funktion `boost::filesystem::last_write_time()` aufgerufen werden.

Beispiel 35.13 Verfügbarer Speicherplatz mit `boost::filesystem::space()` ermitteln

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\"};
  try
  {
    space_info s = space(p);
    std::cout << s.capacity << '\n';
    std::cout << s.free << '\n';
    std::cout << s.available << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Mit der im Beispiel 35.13 aufgerufenen Funktion `boost::filesystem::space()` kann der insgesamt zur Verfügung stehende und momentan noch freie Speicherplatz erhalten werden. Die Funktion gibt ein Objekt vom Typ `boost::filesystem::space_info` zurück, das drei Eigenschaften **`capacity`**, **`free`** und **`available`** definiert. Alle drei Eigenschaften sind vom Typ boost::uintmax_t. Der Speicherplatz wird in Bytes gemessen.
Während die bisher vorgestellten Funktionen Dateien und Verzeichnisse unberührt lassen, existieren zahlreiche Funktionen, mit denen Dateien und Verzeichnisse erstellt, umbenannt oder gelöscht werden können.

Beispiel 35.14 Verzeichnis erstellen, umbenennen und entfernen

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Test"};
  try
  {
    if (create_directory(p))
    {
      rename(p, "C:\\Test2");
      boost::filesystem::remove("C:\\Test2");
    }
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Beispiel 35.14 sollte selbsterklärend sein. Den verschiedenen Funktionen wird jedoch nicht immer ein Objekt vom Typ `boost::filesystem::path` übergeben, sondern teilweise einfach nur ein String. Das ist insofern kein Problem als dass `boost::filesystem::path` einen nicht-expliziten Konstruktor anbietet, so dass Strings automatisch in Objekte vom Typ `boost::filesystem::path` umgewandelt werden. Das macht es einfacher, die vielen Funktionen von Boost.Filesystem zu verwenden, da nicht selbst explizit Objekte vom Typ `boost::filesystem::path` erstellt werden müssen.

> **Anmerkung**
>
> `boost::filesystem::remove()` wird im Beispiel 35.14 explizit über ihren Namensraum aufgerufen, da Visual C++ 2013 die Funktion andernfalls mit `remove()` aus der Headerdatei `stdio.h` verwechselt.

Neben den im Beispiel 35.14 verwendeten Funktionen stehen auch Funktionen wie `create_symlink()` zur Verfügung, mit der sich symbolische Verknüpfungen erstellen lassen, oder `copy_file()`, mit der eine Datei oder ein Verzeichnis kopiert werden kann.
Im Beispiel 35.15 wird Ihnen eine Funktion vorgestellt, die ausgehend von einem Dateinamen oder Teilausschnitt eines Pfads einen absoluten Pfad erstellt.

Beispiel 35.15 Absoluten Pfad mit `boost::filesystem::absolute()` erstellen

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  try
  {
    std::cout << absolute("photo.jpg") << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Welcher Pfad ausgegeben wird, hängt davon ab, in welchem Verzeichnis das Programm gestartet wird. Wird es zum Beispiel in `C:\` ausgeführt, gibt Beispiel 35.15 `"C:\photo.jpg"` aus.
Wenn Sie einen absoluten Pfad relativ zu einem anderen Verzeichnis erhalten möchten, können Sie `boost::filesystem::absolute()` einen zweiten Parameter übergeben.

Beispiel 35.16 Absoluten Pfad relativ zu einem anderen Verzeichnis erstellen

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  try
  {
    std::cout << absolute("photo.jpg", "D:\\") << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Beispiel 35.16 gibt `"D:\photo.jpg"` aus.

Abschließend soll Ihnen eine nützliche Hilfsfunktion vorgestellt werden, mit der sich das aktuelle Arbeitsverzeichnis ermitteln lässt.

Beispiel 35.17 Arbeitsverzeichnis mit `boost::filesystem::current_path()` erhalten

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  try
  {
    std::cout << current_path() << '\n';
    current_path("C:\\");
    std::cout << current_path() << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Beispiel 35.17 ruft mehrfach `boost::filesystem::current_path()` auf. Wird die Funktion ohne Parameter aufgerufen, gibt sie das aktuelle Arbeitsverzeichnis zurück. Wird ihr ein Parameter vom Typ `boost::filesystem::path` übergeben, wird das Arbeitsverzeichnis neu gesetzt.

35.3 Verzeichnisiteratoren

Boost.Filesystem bietet mit `boost::filesystem::directory_iterator` einen Iterator an, um über Dateien in einem Verzeichnis zu iterieren. Sehen Sie sich dazu Beispiel 35.18 an.

Beispiel 35.18 Über Einträge in einem Verzeichnis iterieren

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p = current_path();
  directory_iterator it{p};
  while (it != directory_iterator{})
    std::cout << *it++ << '\n';
}
```

`boost::filesystem::directory_iterator` wird mit einer Pfadangabe initialisiert, um einen Iterator zu erhalten, der auf den Anfang eines Verzeichnisses zeigt. Um das Ende eines Verzeichnisses zu erhalten, muss die Klasse mit dem Standardkonstruktor instanziiert werden.
Sie dürfen während einer Iteration Verzeichniseinträge erstellen oder löschen – der Iterator bleibt gültig. Es ist jedoch nicht definiert, ob Verzeichnisänderungen innerhalb der Iteration sichtbar werden. So zeigt der Iterator unter Umständen nicht auf neu erstellte Dateien. Sie müssen die Iteration neu starten, um sicherzugehen, dass Sie auf alle aktuellen Verzeichniseinträge zugreifen können.
Neben `boost::filesystem::directory_iterator` stellt Boost.Filesystem mit `boost::filesystem::recursive_directory_iterator` einen Iterator zur Verfügung, um rekursiv über Verzeichnisse zu iterieren.

35.4 Dateistreams

Im Standard sind verschiedene Dateistreams in der Headerdatei `fstream` definiert. Diese Streams akzeptieren keine Parameter vom Typ `boost::filesystem::path`. Möchten Sie Dateistreams mit Pfadangaben vom Typ

`boost::filesystem::path` öffnen, kann Sie dazu auf die Headerdatei `boost/filesystem/fstream.hpp` zugreifen.

Beispiel 35.19 `boost::filesystem::ofstream` in Aktion

```
#include <boost/filesystem/fstream.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"test.txt"};
  ofstream ofs{p};
  ofs << "Hello, world!\n";
}
```

Im Beispiel 35.19 wird auf eine Datei mit Hilfe der Klasse `boost::filesystem::ofstream` zugegriffen. Dem Konstruktor wird ein Objekt vom Typ `boost::filesystem::path` übergeben. Selbstverständlich ist es auch möglich, ein Objekt vom Typ `boost::filesystem::path` an `open()` zu übergeben.

Teil VIII

Zeitangaben

Die folgenden Bibliotheken können Sie zum Verarbeiten von Zeitangaben verwenden.

- Boost.DateTime definiert zahlreiche Klassen für Zeitpunkte und Perioden – sowohl für Uhrzeiten als auch für Kalenderdaten – als auch Funktionen, um diese zu verarbeiten. So können Sie zum Beispiel über Kalenderdaten iterieren.
- Boost.Chrono und Boost.Timer stellen Uhren zur Verfügung, um Zeit zu messen. Die von Boost.Timer angebotenen Uhren sind darauf spezialisiert, die Ausführungsgeschwindigkeit von Code zu messen. Boost.Timer wird also lediglich zur Optimierung von Code verwendet.

Kapitel 36

Boost.DateTime

Die Bibliothek Boost.DateTime kann verwendet werden, um Zeitangaben wie Kalenderdaten und Uhrzeiten zu verarbeiten. Darüber hinaus bietet Boost.DateTime Erweiterungen an, um zum Beispiel Zeitzonen zu berücksichtigen. Außerdem wird die formatierte Ein- und Ausgabe von Kalenderdaten und Uhrzeiten unterstützt. Sind Sie eher daran interessiert, die aktuelle Uhrzeit zu ermitteln und abhängig von dieser Zeiten zu messen, sehen Sie sich Boost.Chrono im Kapitel 37 an.

36.1 Kalenderdaten

Boost.DateTime unterstützt von Haus aus lediglich Kalenderdaten basierend auf dem *Gregorianischen Kalender*. Das ist in der Praxis insofern kein Problem, als dass der Gregorianische Kalender der heute weltweit am weitesten verbreitete Kalender ist. So können Sie davon ausgehen, dass Sie, wenn Sie sich mit jemandem zum Beispiel für den 12. Mai 2014 verabreden, ihm nicht zusätzlich mitteilen müssen, dass sich das Datum auf den Gregorianischen Kalender bezieht.
Der Gregorianische Kalender wurde von Papst Gregor XIII. im Jahr 1582 eingeführt. Das heißt, dass erst seit diesem Jahr mit dem Gregorianischen Kalender gerechnet wird. Das ist insofern wichtig, als dass Boost.DateTime wie erwähnt ausschließlich den Gregorianischen Kalender unterstützt. Genaugenommen werden Datumsangaben für die Jahre 1400 bis 9999 unterstützt. Die Unterstützung reicht demnach über das Jahr 1582 hinaus bis zurück ins Jahr 1400. Falls Sie mit Datumsangaben vor 1582 arbeiten, können Sie Boost.DateTime verwenden, wenn Sie die Datumsangaben in den Gregorianischen Kalender umrechnen und nur bis ins Jahr 1400 zurückgehen müssen. Es besteht jedoch auch die Möglichkeit, Boost.DateTime um einen neuen Kalendar zu erweitern.
Der Namensraum, in dem Boost.DateTime Klassen und Funktionen zur Verarbeitung von Kalenderdaten zur Verfügung stellt, ist `boost::gregorian`. Sie können alle Klassen und Funktionen aus diesem Namensraum einsetzen, wenn Sie die Headerdatei `boost/date_time/gregorian/gregorian.hpp` einbinden. So können Sie dann auf die Klasse `boost::gregorian::date` zugreifen, um ein Datum zu erstellen.

Beispiel 36.1 Ein Datum mit `boost::gregorian::date` erstellen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d{2014, 1, 31};
  std::cout << d.year() << '\n';
  std::cout << d.month() << '\n';
  std::cout << d.day() << '\n';
  std::cout << d.day_of_week() << '\n';
  std::cout << d.end_of_month() << '\n';
}
```

Die Klasse `boost::gregorian::date` bietet mehrere Konstruktoren an, um ein Datum zu erstellen. Dem einfachsten Konstruktor wird ein Jahr, Monat und Tag übergeben. Wird ein ungültiger Wert angegeben, wird eine Ausnahme vom Typ `boost::gregorian::bad_year`, `boost::gregorian::bad_month` oder `boost::gregorian::bad_day_of_month` geworfen, die alle von `std::out_of_range` abgeleitet sind.

Wie Sie am Beispiel 36.1 sehen, stehen zahlreiche Methoden zur Verfügung, um auf ein Datum zuzugreifen. Während über Methoden wie `year()`, `month()` und `day()` auf die Werte zugegriffen werden kann, mit denen das Objekt initialisiert wurde, können mit Methoden wie `day_of_week()` und `end_of_month()` Werte erhalten werden, die selbst zu errechnen aufwändig wäre.
Während Sie dem Konstruktor von `boost::gregorian::date` wie im Beispiel 36.1 Zahlen übergeben können, um ein Datum zu setzen, wird beim Aufruf von `month()` `Jan` und beim Aufruf von `day_of_week()` `Fri` ausgegeben. Es handelt sich dabei um Werte vom Typ boost::gregorian::date::month_type und boost::gregorian::date::day_of_w
Wie Sie später in diesem Kapitel noch sehen werden, bietet Boost.DateTime eine umfangreiche Unterstützung zur formatierten Ein- und Ausgabe an, so dass Sie zum Beispiel die Ausgabe so anpassen können, dass anstatt von `Jan` der Wert `1` ausgegeben wird.
Beachten Sie, dass der Standardkonstruktor von `boost::gregorian::date` ein ungültiges Datum erstellt. Sie können ein derart ungültiges Datum explizit erstellen, wenn Sie `boost::date_time::not_a_date_time` als einzigen Parameter an den Konstruktor von `boost::gregorian::date` übergeben.
Neben dem direkten Aufruf eines Konstruktors kann ein Objekt vom Typ `boost::gregorian::date` auch über freistehende Funktionen und Methoden anderer Objekte erstellt werden.

Beispiel 36.2 Ein Datum von einer Uhr oder einem String erhalten

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d = day_clock::universal_day();
  std::cout << d.year() << '\n';
  std::cout << d.month() << '\n';
  std::cout << d.day() << '\n';

  d = date_from_iso_string("20140131");
  std::cout << d.year() << '\n';
  std::cout << d.month() << '\n';
  std::cout << d.day() << '\n';
}
```

Im Beispiel 36.2 wird die Klasse `boost::gregorian::day_clock` verwendet. Es handelt sich dabei um eine Uhr, die das aktuelle Datum zurückgibt. Die Methode `universal_day()` gibt ein UTC-Datum zurück, das unabhängig von Zeitzonen oder Sommerzeit ist. UTC ist die internationale Abkürzung für die Weltzeit, der Mitteleuropa eine Stunde voraus ist. Neben `universal_day()` bietet `boost::gregorian::day_clock` auch eine Methode `local_day()` an, die die Systemeinstellungen bezüglich Zeitzone und Sommerzeit berücksichtigt. Sie verwenden `local_day()`, wenn Sie ein aktuelles Datum in der Zeitzone erhalten wollen, in der Sie sich befinden.
Im Namensraum `boost::gregorian` stehen außerdem zahlreiche freistehende Funktionen zur Verfügung, um ein Datum in einem String in ein Objekt vom Typ `boost::gregorian::date` umzuwandeln. So wird im Beispiel 36.2 mit der Funktion `boost::gregorian::date_from_iso_string()` ein Datum im ISO 8601-Format umgewandelt. Boost.DateTime bietet weitere freistehende Funktionen an wie `boost::gregorian::from_simple_string()` und `boost::gregorian::from_us_string()`.
Während Sie mit `boost::gregorian::date` eine Klasse kennengelernt haben, die als Datum einen Zeitpunkt markiert, bietet Boost.DateTime mit `boost::gregorian::date_duration` eine Klasse für einen Zeitraum an.

Beispiel 36.3 Zeiträume mit `boost::gregorian::date_duration` erstellen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d1{2014, 1, 31};
  date d2{2014, 2, 28};
  date_duration dd = d2 - d1;
  std::cout << dd.days() << '\n';
```

```
}
```

Da die Klasse `boost::gregorian::date` den Operator `operator-` überlädt, können wie im Beispiel 36.3 zwei Zeitpunkte mit dem Minuszeichen verknüpft werden. Der Rückgabewert hat den Typ `boost::gregorian::date_duration` und entspricht dem Zeitraum zwischen den beiden Daten.
Die wichtigste Methode, die `boost::gregorian::date_duration` anbietet, ist `days()`. Sie gibt die Anzahl der Tage zurück, aus denen der Zeitraum besteht.

Beispiel 36.4 Spezialisierte Zeiträume

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date_duration dd{4};
  std::cout << dd.days() << '\n';
  weeks ws{4};
  std::cout << ws.days() << '\n';
  months ms{4};
  std::cout << ms.number_of_months() << '\n';
  years ys{4};
  std::cout << ys.number_of_years() << '\n';
}
```

Sie können ein Objekt vom Typ `boost::gregorian::date_duration` auch selbst erstellen. Sie übergeben in diesem Fall dem Konstruktor als einzigen Parameter die Anzahl der Tage. Wenn Sie einen Zeitraum erstellen wollen, der Wochen, Monate oder Jahre umfasst, können Sie wie im Beispiel 36.4 auf die Klassen `boost::gregorian::weeks`, `boost::gregorian::months` und `boost::gregorian::years` zugreifen.
Die Klassen `boost::gregorian::months` und `boost::gregorian::years` bieten keine Methoden an, um die Anzahl der Tage zu ermitteln – immerhin können Monate und Jahre unterschiedlich lang sein. Warum es dennoch sinnvoll sein kann, diese Klassen zu verwenden, sehen Sie am Beispiel 36.5.

Beispiel 36.5 Mit spezialisierten Zeiträumen rechnen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d{2014, 1, 31};
  months ms{1};
  date d2 = d + ms;
  std::cout << d2 << '\n';
  date d3 = d2 - ms;
  std::cout << d3 << '\n';
}
```

Im Beispiel 36.5 wird zum 31. Januar 2014 ein Monat hinzuaddiert. Boost.DateTime errechnet für **`d2`** daraufhin den 28. Februar 2014. Im nächsten Schritt wird von diesem Datum ein Monat abgezogen, woraufhin für **`d3`** wieder der 31. Januar 2014 errechnet wird. Wie Sie sehen, lassen sich mit Zeitpunkten und -räumen Berechnungen anstellen. Sie müssen jedoch auf Besonderheiten achten wie die, dass Sie ausgehend vom letzten Tag eines Monats immer zum letzten Tag eines anderen Monats gelangen, wenn Sie mit `boost::gregorian::months` vorwärts- oder zurückspringen. Das kann zu unerwarteten Ergebnissen führen.

Beispiel 36.6 Besonderheiten beim Rechnen mit spezialisierten Zeiträumen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;
```

```
int main()
{
  date d{2014, 1, 30};
  months ms{1};
  date d2 = d + ms;
  std::cout << d2 << '\n';
  date d3 = d2 - ms;
  std::cout << d3 << '\n';
}
```

Beispiel 36.6 ähnelt dem vorherigen. Der einzige Unterschied ist, dass die Variable **d** mit dem 30. Januar 2014 initialisiert wurde. Obwohl es sich hierbei nicht um den letzten Tag im Januar handelt, wird nach dem Sprung vorwärts für **d2** wieder der 28. Februar 2014 errechnet – schließlich gibt es keinen 30. Februar. Wenn im letzten Schritt ein Monat zurückgesprungen wird, wird für **d3** jedoch der 31. Januar 2014 errechnet. Schließlich ist der 28. Februar 2014 der letzte Tag in diesem Monat, so dass Sie bei einem Sprung zurück zum letzten Tag im Januar gelangen.
Wenn Sie das für verwirrend halten, können Sie die Definition des Makros `BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES` aufheben. Sie können dann die Klassen `boost::gregorian::weeks`, `boost::gregorian::months` und `boost::gregorian::years` nicht verwenden. Da dann lediglich die Klasse `boost::gregorian::date_duration` zur Verfügung steht, mit der eine bestimmte Anzahl an Tagen vorwärts oder rückwärts gezählt wird, kann es nicht mehr zu möglicherweise unerwarteten Ergebnissen kommen.

Beispiel 36.7 Zeitintervalle mit `boost::gregorian::date_period` erstellen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d1{2014, 1, 1};
  date d2{2014, 2, 28};
  date_period dp{d1, d2};
  date_duration dd = dp.length();
  std::cout << dd.days() << '\n';
}
```

Während Sie mit `boost::gregorian::date_duration` eine Klasse kennengelernt haben, die einen beliebigen Zeitraum umfasst, können Sie mit `boost::gregorian::date_period` ein Zeitintervall angeben, das an einem bestimmten Datum beginnt und an einem bestimmten Datum endet.
Dem Konstruktor von `boost::gregorian::date_period` werden wie im Beispiel 36.7 zwei Parameter vom Typ `boost::gregorian::date` übergeben, die den Anfangs- und Endzeitpunkt bestimmen. Alternativ kann auch ein Anfangszeitpunkt und ein Zeitraum vom Typ `boost::gregorian::date_duration` angegeben werden. Beachten Sie, dass in beiden Fällen der Endzeitpunkt nicht zum Zeitintervall zählt, sondern der Tag vor dem Endzeitpunkt der letzte Tag im Zeitintervall ist. Das ist wichtig, um zu verstehen, welche Ergebnisse Beispiel 36.8 ausgibt.

Beispiel 36.8 Überprüfen, ob Daten in einem Zeitintervall liegen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d1{2014, 1, 1};
  date d2{2014, 2, 28};
  date_period dp{d1, d2};
  std::cout.setf(std::ios::boolalpha);
  std::cout << dp.contains(d1) << '\n';
  std::cout << dp.contains(d2) << '\n';
}
```

Im Beispiel 36.8 wird mit Hilfe der Methode `contains()` getestet, ob ein bestimmes Datum innerhalb des Zeitintervalls liegt. Obwohl dem Konstruktor von `boost::gregorian::date_period` die beiden Variablen **`d1`** und **`d2`** übergeben wurden, gibt `contains()` lediglich beim ersten Aufruf `true` zurück. Weil der Endzeitpunkt nicht zum Zeitintervall zählt, wird beim Aufruf von `contains()` mit **`d2`** als Parameter `false` zurückgegeben.
Die Klasse `boost::gregorian::date_period` bietet weitere Methoden an, um zum Beispiel ein Zeitintervall zu verschieben oder die Schnittmenge zweier sich überschneidender Zeitintervalle zu ermitteln.
Neben Klassen für Daten, Zeiträume und Zeitintervalle bietet Boost.DateTime Iteratoren und verschiedene nützliche freistehende Funktionen an. Sehen Sie sich dazu Beispiel 36.9 an.

Beispiel 36.9 Mit Iteratoren auf Datumsangaben zugreifen

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost;

int main()
{
  gregorian::date d{2014, 5, 12};
  gregorian::day_iterator it{d};
  std::cout << *++it << '\n';
  std::cout << date_time::next_weekday(*it,
    gregorian::greg_weekday(date_time::Friday)) << '\n';
}
```

Der Iterator `boost::gregorian::day_iterator` ermöglicht es, von einem bestimmten Datum ausgehend schrittweise einen Tag vorwärts oder rückwärts zu zählen. Neben diesem Iterator werden auch `boost::gregorian::week_iterator`, `boost::gregorian::month_iterator` und `boost::gregorian::year_iterator` angeboten, die jeweils Wochen, Monate oder Jahre vor- oder zurückspringen.
Im Beispiel 36.9 wird außerdem die Funktion `boost::date_time::next_weekday()` verwendet, die das Datum des nächsten Wochentags ausgehend von einem Zeitpunkt zurückgibt. So gibt Beispiel 36.9 `2014-May-16` aus, weil dies das Datum für den ersten Freitag nach dem 13. Mai 2014 ist.

36.2 Ortsunabhängige Zeitpunkte

Während mit `boost::gregorian::date` ein Datum erstellt werden kann, kann `boost::posix_time::ptime` verwendet werden, um einen ortsunabhängigen Zeitpunkt zu definieren. `boost::posix_time::ptime` greift hierzu auf `boost::gregorian::date` zu, speichert darüber hinaus aber eine Uhrzeit.
Um `boost::posix_time::ptime` verwenden zu können, müssen Sie die Headerdatei `boost/date_time/posix_time/posix_time.hpp` einbinden.

Beispiel 36.10 `boost::posix_time::ptime` in Aktion

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  date d = pt.date();
  std::cout << d << '\n';
  time_duration td = pt.time_of_day();
  std::cout << td << '\n';
}
```

Sie initialisieren ein Objekt vom Typ `boost::posix_time::ptime`, indem Sie dem Konstruktor als ersten Parameter ein Datum vom Typ `boost::gregorian::date` und als zweiten Parameter einen Zeitraum vom Typ

`boost::posix_time::time_duration` übergeben. Die drei Parameter, die dem Konstruktor von `boost::posix_time::time_duration` übergeben werden, bestimmen die Uhrzeit. Im Beispiel 36.10 wird ein Zeitpunkt für exakt 12 Uhr mittags am 12. Mai 2014 definiert.
Über die beiden Methoden `date()` und `time_of_day()`, die im Beispiel 36.10 verwendet werden, können Sie das Datum und die Uhrzeit abfragen.
So wie der Standardkonstruktor von `boost::gregorian::date` ein ungültiges Datum erstellt, ist ein Zeitpunkt vom Typ `boost::posix_time::ptime` ungültig, wenn der Standardkonstruktor ausgeführt wird. Sie können auch explizit einen ungültigen Zeitpunkt erstellen, indem Sie `boost::date_time::not_a_date_time` als einzigen Parameter an den Konstruktor von `boost::posix_time::ptime` übergeben.
So wie Kalenderdaten vom Typ `boost::gregorian::date` über den Aufruf von Methoden anderer Objekte und freistehender Funktionen erzeugt werden können, bietet Boost.DateTime entsprechende Objekte und freistehende Funktionen zum Erzeugen von Zeitpunkten an.

Beispiel 36.11 Einen Zeitpunkt von einer Uhr oder einem String erhalten

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::posix_time;

int main()
{
  ptime pt = second_clock::universal_time();
  std::cout << pt.date() << '\n';
  std::cout << pt.time_of_day() << '\n';

  pt = from_iso_string("20140512T120000");
  std::cout << pt.date() << '\n';
  std::cout << pt.time_of_day() << '\n';
}
```

Die Klasse `boost::posix_time::second_clock` repräsentiert eine Uhr, die die aktuelle Zeit zurückgibt. Die Methode `universal_time()`, die im Beispiel 36.11 aufgerufen wird, gibt die UTC-Uhrzeit zurück. Wenn Sie nicht die Weltzeit verwenden möchten, sondern die aktuelle Zeit in der Zeitzone, in der Sie sich befinden, rufen Sie `local_time()` auf.
Boost.DateTime bietet eine weitere Klasse `boost::posix_time::microsec_clock` an, die die aktuelle Uhrzeit einschließlich Mikrosekunden zurückgibt. Wenn Sie eine höhere Auflösung benötigen, können Sie diese Klasse verwenden.
Freistehende Funktionen wie `boost::posix_time::from_iso_string()` können verwendet werden, um einen Zeitpunkt von einem String in ein Objekt vom Typ `boost::posix_time::ptime` umzuwandeln. Die Funktion `boost::posix_time::from_iso_string()` setzt dabei voraus, dass der Zeitpunkt im String im ISO 8601-Format gespeichert ist.

Beispiel 36.12 Zeiträume mit `boost::posix_time::time_duration` erstellen

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;

int main()
{
  time_duration td{16, 30, 0};
  std::cout << td.hours() << '\n';
  std::cout << td.minutes() << '\n';
  std::cout << td.seconds() << '\n';
  std::cout << td.total_seconds() << '\n';
}
```

Neben `boost::posix_time::ptime` bietet Boost.DateTime auch eine Klasse `boost::posix_time::time_duration` an, um einen Zeitraum zu beschreiben. Sie haben diese Klasse bereits gesehen, da der Konstruktor von `boost::posix_time::ptime` als zweiten Parameter ein Objekt vom Typ `boost::posix_time::time_`

duration erwartet. Sie können die Klasse boost::posix_time::time_duration aber auch ohne boost::posix_time::ptime verwenden.
Während Sie mit hours(), minutes() und seconds() die Werte erhalten, die als Parameter dem Konstruktor übergeben wurden, können Sie mit Methoden wie total_seconds() auf einfache Weise zusätzliche Informationen erhalten. So gibt Ihnen total_seconds() die Anzahl aller Sekunden zurück.
Sie können dem Konstruktor von boost::posix_time::time_duration beliebige Werte übergeben. Es gibt also zum Beispiel keine Obergrenze von 24 Stunden.
So wie mit Kalenderdaten kann auch mit Zeitpunkten und Zeiträumen gerechnet werden.

Beispiel 36.13 Mit Zeitpunkten rechnen

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt1{date{2014, 5, 12}, time_duration{12, 0, 0}};
  ptime pt2{date{2014, 5, 12}, time_duration{18, 30, 0}};
  time_duration td = pt2 - pt1;
  std::cout << td.hours() << '\n';
  std::cout << td.minutes() << '\n';
  std::cout << td.seconds() << '\n';
}
```

Wenn wie im Beispiel 36.13 zwei Zeitpunkte vom Typ boost::posix_time::ptime mit einem Minuszeichen verknüpft werden, erhalten Sie ein Objekt vom Typ boost::posix_time::time_duration, das den Zeitraum zwischen den beiden Zeitpunkten angibt.

Beispiel 36.14 Mit Zeiträumen rechnen

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt1{date{2014, 5, 12}, time_duration{12, 0, 0}};
  time_duration td{6, 30, 0};
  ptime pt2 = pt1 + td;
  std::cout << pt2.time_of_day() << '\n';
}
```

Wie Sie am Beispiel 36.14 sehen, kann ein Zeitraum mit dem Pluszeichen auch zu einem Zeitpunkt hinzugefügt werden. In diesem Fall wird ein neuer Zeitpunkt erhalten. Obiges Programm gibt 18:30:00 auf die Standardausgabe aus.
Boost.DateTime verwendet für Kalenderdaten und beliebige Zeitpunkte die gleichen Konzepte. So wie es jeweils eine Klasse gibt, die einen Zeitpunkt und einen Zeitraum beschreibt, gibt es auch eine Klasse, die ein Zeitintervall repräsentiert. Für Kalenderdaten heißt diese Klasse boost::gregorian::date_period, für beliebige Zeitpunkte boost::posix_time::time_period. Und so wie der Konstruktor von boost::gregorian::date_period als Parameter zwei Kalenderdaten erwartet, müssen dem Konstruktor von boost::posix_time::time_period zwei Zeitpunkte übergeben werden.

Beispiel 36.15 Zeitintervalle mit boost::posix_time::time_period erstellen

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
```

```
{
  ptime pt1{date{2014, 5, 12}, time_duration{12, 0, 0}};
  ptime pt2{date{2014, 5, 12}, time_duration{18, 30, 0}};
  time_period tp{pt1, pt2};
  std::cout.setf(std::ios::boolalpha);
  std::cout << tp.contains(pt1) << '\n';
  std::cout << tp.contains(pt2) << '\n';
}
```

Die Klasse `boost::posix_time::time_period` funktioniert genauso wie `boost::gregorian::date_period`. So wird unter anderem eine Methode `contains()` angeboten, die für jeden Zeitpunkt, der sich innerhalb des Zeitintervalls befindet, `true` zurückgibt. Da der Zeitpunkt, der dem Konstruktor von `boost::posix_time::time_period` als zweiter Parameter übergeben wurde, gerade nicht mehr zum Zeitintervall zählt, gibt Beispiel 36.15 beim zweiten Aufruf von `contains()` `false` zurück.
Neben `contains()` bietet `boost::posix_time::time_period` weitere Methoden an wie `intersection()`, um eine Schnittmenge zweier Zeitintervalle zu ermitteln, oder `merge()`, um zwei Zeitintervalle, die sich überschneiden, zu einem neuen Zeitintervall zu vereinigen.
Abschließend soll der Iterator `boost::posix_time::time_iterator` vorgestellt werden, mit dem über Zeitpunkte iteriert werden kann.

Beispiel 36.16 Mit Iteratoren auf Zeitpunkte zugreifen

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  time_iterator it{pt, time_duration{6, 30, 0}};
  std::cout << *++it << '\n';
  std::cout << *++it << '\n';
}
```

Im Beispiel 36.16 wird der Iterator **`it`** verwendet, um ausgehend vom Zeitpunkt **`pt`** jeweils sechseinhalb Stunden vorwärts zu springen. Da der Iterator zweimal inkrementiert wird, gibt das Programm entsprechend `2014-May-12 18:30:00` und `2014-May-13 01:00:00` aus.

36.3 Ortsabhängige Zeitpunkte

Der Unterschied zu den im vorherigen Abschnitt kennengelernten ortsunabhängigen Zeitpunkten ist, dass ortsabhängige Zeitpunkte Zeitzonen berücksichtigen. Dazu bietet Boost.DateTime eine Klasse `boost::local_time::local_date_time` an, die die Klasse `boost::local_time::posix_time_zone` verwendet, in der Einstellungen zur Zeitzone gespeichert sind. Um auf diese Klassen zugreifen zu können, muss die Headerdatei `boost/date_time/local_time/local_time.hpp` eingebunden werden.

Beispiel 36.17 `boost::local_time::local_date_time` in Aktion

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::local_time;
using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  time_zone_ptr tz{new posix_time_zone{"CET+1"}};
  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  local_date_time dt{pt, tz};
  std::cout << dt.utc_time() << '\n';
```

```
  std::cout << dt << '\n';
  std::cout << dt.local_time() << '\n';
  std::cout << dt.zone_name() << '\n';
}
```

Der Konstruktor der Klasse `boost::local_time::local_date_time` erwartet als ersten Parameter ein Objekt vom Typ `boost::posix_time::ptime` und als zweiten Parameter ein Objekt vom Typ boost::local_time::time_zone_ptr. Es handelt sich dabei um eine Typdefinition für boost::shared_ptr<boost::local_time::time_zone>. Die Typdefinition verwendet `boost::local_time::time_zone` und nicht `boost::local_time::posix_time_zone`. Das ist insofern kein Problem, als dass `boost::local_time::posix_time_zone` von `boost::local_time::time_zone` abgeleitet ist. Das macht es möglich, Boost.DateTime um benutzerdefinierte Zeitzonen zu erweitern.
An den Konstruktor von `boost::local_time::local_date_time` wird demnach kein Objekt vom Typ `boost::local_time::posix_time_zone` übergeben, sondern ein Smartpointer auf dieses. So können sich mehrere Objekte vom Typ `boost::local_time::local_date_time` eine Zeitzone teilen. Wenn das letzte Objekt zerstört wird, wird automatisch das entsprechende Objekt, das die Zeitzone repräsentiert, freigegeben.
Um ein Objekt vom Typ `boost::local_time::posix_time_zone` zu erstellen, wird dem Konstruktor als einziger Parameter ein String übergeben, der die Zeitzone beschreibt. Im Beispiel 36.17 wird angegeben, dass es sich bei der Zeitzone um Mitteleuropa handelt: CET ist die Abkürzung für Central European Time. Da CET eine Stunde vor UTC liegt, wird als Abweichung in Stunden +1 angegeben. Boost.DateTime interpretiert Abkürzungen für Zeitzonen nicht selbst und weiß nicht, was CET bedeutet. Sie müssen immer eine Abweichung in Stunden angeben. Wollen Sie keine Abweichung verwenden, geben Sie +0 an.
Wenn Sie Beispiel 36.17 ausführen, wird `2014-May-12 12:00:00`, `2014-May-12 13:00:00 CET`, `2014-May-12 13:00:00` und `CET` auf die Standardausgabe ausgegeben. Werte, mit denen Objekte vom Typ `boost::posix_time::ptime` und `boost::local_time::local_date_time` initialisiert werden, werden grundsätzlich auf die UTC-Zeitzone bezogen. Erst bei der Ausgabe eines Objekts vom Typ `boost::local_time::local_date_time` auf die Standardausgabe und beim Aufruf von `local_time()` wird die Abweichung in Stunden zur Berechnung der lokalen Uhrzeit herangezogen.

Beispiel 36.18 Ortsabhängige Zeitpunkte und unterschiedliche Zeitzonen

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::local_time;
using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  time_zone_ptr tz{new posix_time_zone{"CET+1"}};

  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  local_date_time dt{pt, tz};
  std::cout << dt.local_time() << '\n';

  time_zone_ptr tz2{new posix_time_zone{"EET+2"}};
  std::cout << dt.local_time_in(tz2).local_time() << '\n';
}
```

Wenn Sie die Methode `local_time()` verwenden, wird die Abweichung in der Zeitzone berücksichtigt. Wenn der UTC-Zeitpunkt 12 Uhr mittags in **`dt`** für die Zeitzone CET berechnet werden soll, bedeutet das, dass eine Stunde hinzugefügt werden muss. Denn CET ist der Zeitzone UTC um eine Stunde voraus. Die Methode `local_time()` gibt demnach `2014-May-12 13:00:00` zurück.
Die Methode `local_time_in()` interpretiert den Zeitpunkt in **`dt`** so, als läge er in der Zeitzone, die als Parameter übergeben wird. Für Beispiel 36.18 bedeutet dies, dass 12 Uhr mittags in UTC 14 Uhr in EET ergibt. EET ist die Abkürzung für Eastern European Time, die um zwei Stunden vor UTC liegt.
Nachdem Sie ortsabhängige Zeitpunkte kennengelernt haben, wird Ihnen abschließend das ortsabhängige Zeitintervall vorgestellt. Dazu stellt Boost.DateTime die Klasse `boost::local_time::local_time_period` zur Verfügung.

Beispiel 36.19 `boost::local_time::local_time_period` in Aktion

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::local_time;
using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  time_zone_ptr tz{new posix_time_zone{"CET+0"}};

  ptime pt1{date{2014, 12, 5}, time_duration{12, 0, 0}};
  local_date_time dt1{pt1, tz};

  ptime pt2{date{2014, 12, 5}, time_duration{18, 0, 0}};
  local_date_time dt2{pt2, tz};

  local_time_period tp{dt1, dt2};

  std::cout.setf(std::ios::boolalpha);
  std::cout << tp.contains(dt1) << '\n';
  std::cout << tp.contains(dt2) << '\n';
}
```

Wie im Beispiel 36.19 zu sehen, erwartet der Konstruktor von `boost::local_time::local_time_period` zwei Parameter vom Typ `boost::local_time::local_date_time`. So wie bei den anderen Typen für Zeitintervalle, die Boost.DateTime anbietet, ist der zweite Parameter, der den Endzeitpunkt darstellt, gerade nicht mehr im Zeitintervall enthalten. Über die in diesem Kapitel mehrfach erwähnten Methoden `contains()`, `intersection()`, `merge()` und andere können auch bei `boost::local_time::local_time_period` Zeitintervalle verarbeitet werden.

36.4 Formatierte Ein- und Ausgabe

Wenn Sie die Beispiele in diesem Kapitel ausgeführt haben, haben Sie Ergebnisse wie `2014-May-12` erhalten. Möglicherweise ziehen Sie anders formatierte Ergebnisse vor. Boost.DateTime bietet mit `boost::date_time::date_facet` und `boost::date_time::time_facet` Klassen an, mit denen Sie Kalenderdaten und Zeitpunkte formatieren können.
Boost.DateTime greift auf das aus dem Standard bekannte Konzept der Locales zu. Um mit Boost.DateTime ein Kalenderdatum zu formatieren, müssen Sie ein Objekt vom Typ `boost::date_time::date_facet` erstellen und es in einem Locale installieren. Dem Konstruktor von `boost::date_time::date_facet` übergeben Sie einen String, der das neue Format beschreibt. Im Beispiel 36.20 ist „%A, %d %B %Y“ angegeben, was bedeutet, dass der Wochentag gefolgt vom Datum mit ausgeschriebenem Monatsnamen ausgegeben wird: `Monday, 12 May 2014`.

Beispiel 36.20 Benutzerdefinierte Formatierung eines Datums

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>
#include <locale>

using namespace boost::gregorian;

int main()
{
  date d{2014, 5, 12};
  date_facet *df = new date_facet{"%A, %d %B %Y"};
  std::cout.imbue(std::locale{std::cout.getloc(), df});
  std::cout << d << '\n';
}
```

Boost.DateTime bietet zahlreiche Formatierungsflags an, die jeweils aus einem Prozentzeichen gefolgt von einem Buchstaben bestehen. Die Dokumentation von Boost.DateTime enthält eine vollständige Übersicht über alle unterstützten Formatierungsflags. So ist dort unter anderem angegeben, dass mit „%A“ der Name des Wochentags ausgegeben wird.
Wenn Ihr Programm von Anwendern in Deutschland oder anderen deutschsprachigen Ländern verwendet wird, ziehen Sie es womöglich vor, wenn sowohl der Wochentag als auch der Monat in Deutsch ausgegeben wird.

Beispiel 36.21 Wochen- und Monatsnamen ändern

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <string>
#include <vector>
#include <locale>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  std::locale::global(std::locale{"German"});
  std::string months[12]{"Januar", "Februar", "M\xe4rz", "April",
    "Mai", "Juni", "Juli", "August", "September", "Oktober",
    "November", "Dezember"};
  std::string weekdays[7]{"Sonntag", "Montag", "Dienstag",
    "Mittwoch", "Donnerstag", "Freitag", "Samstag"};
  date d{2014, 5, 12};
  date_facet *df = new date_facet{"%A, %d. %B %Y"};
  df->long_month_names(std::vector<std::string>{months, months + 12});
  df->long_weekday_names(std::vector<std::string>{weekdays,
    weekdays + 7});
  std::cout.imbue(std::locale{std::cout.getloc(), df});
  std::cout << d << '\n';
}
```

Sie können die Namen von Wochentagen und Monaten ändern, indem Sie Vektoren mit den entsprechenden Namen an die Methoden `long_month_names()` und `long_weekday_names()` der Klasse `boost::date_time::date_facet` übergeben. Beispiel 36.21 gibt daraufhin `Montag, 12.Mai 2014` aus.

> **Anmerkung**
>
> Ersetzen Sie die Angabe „German“ mit „de_DE“, wenn Sie das Beispiel auf einem POSIX-Betriebssystem ausführen möchten. Stellen Sie außerdem sicher, dass das Locale für den deutschen Sprach- und Kulturkreis installiert ist.

Boost.DateTime ist sehr flexibel, was die formatierte Datenein- und -ausgabe betrifft. So gibt es neben den beiden Klassen `boost::date_time::date_facet` und `boost::date_time::time_facet` zur Datenausgabe die Klassen `boost::date_time::date_input_facet` und `boost::date_time::time_input_facet` zur formatierten Dateneingabe. Alle vier Klassen bieten zahlreiche Methoden an, mit denen die Ein- und Ausgabe verschiedener Objekte aus Boost.DateTime konfiguriert werden kann. So ist es zum Beispiel auch möglich anzugeben, wie Zeitintervalle vom Typ `boost::gregorian::date_period` ein- und ausgegeben werden können. Aufgrund der vielfältigen Möglichkeiten zur formatierten Ein- und Ausgabe ist ein Blick in die Dokumentation von Bibliothek unerlässlich.

Kapitel 37

Boost.Chrono

Die Bibliothek Boost.Chrono bietet verschiedene Uhren an. So können Sie zum Beispiel die aktuelle Uhrzeit erhalten, so wie sie üblicherweise auf dem Desktop Ihres Computers angezeigt wird. Sie können jedoch auch Uhren verwenden, die es Ihnen erlauben zu messen, wie viel Zeit in einem Prozess verstrichen ist.
Boost.Chrono ist teilweise in den Standard C++11 eingegangen. So können Sie, wenn Ihre Entwicklungsumgebung C++11 unterstützt, auf einige Uhren über die Headerdatei `chrono` zugreifen. C++11 bietet aber zum Beispiel keine Uhren an, um die CPU-Zeit zu messen. Boost.Chrono unterstützt außerdem eine benutzerdefinierte Formatierung zur Ausgabe von Zeiten.
Sie können auf alle Uhren aus Boost.Chrono über die Headerdatei `boost/chrono.hpp` zugreifen. Einzige Ausnahme ist die benutzerdefinierte Formatierung. In diesem Fall müssen Sie zusätzlich die Headerdatei `boost/chrono_io.hpp` einbinden.

Beispiel 37.1 Alle Uhren von Boost.Chrono

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << system_clock::now() << '\n';
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
  std::cout << steady_clock::now() << '\n';
#endif
  std::cout << high_resolution_clock::now() << '\n';

#ifdef BOOST_CHRONO_HAS_PROCESS_CLOCKS
  std::cout << process_real_cpu_clock::now() << '\n';
  std::cout << process_user_cpu_clock::now() << '\n';
  std::cout << process_system_cpu_clock::now() << '\n';
  std::cout << process_cpu_clock::now() << '\n';
#endif

#ifdef BOOST_CHRONO_HAS_THREAD_CLOCK
  std::cout << thread_clock::now() << '\n';
#endif
}
```

Beispiel 37.1 stellt alle Uhren vor, die Boost.Chrono anbietet. Allen Uhren ist gemeinsam, dass sie eine Methode `now()` definieren, die einen Zeitpunkt zurückgibt. Alle Zeitpunkte werden abhängig von der Uhr relativ zu einem universell gültigen Zeitpunkt gemessen. Man spricht dabei von einer *Epoche*. Eine oft verwendete Epoche ist der 1. Januar 1970. Wenn Sie Beispiel 37.1 ausführen, wird für alle Zeitpunkte die Epoche mit ausgegeben.
Boost.Chrono kennt folgende Uhren:

- `boost::chrono::system_clock` gibt die Systemzeit an. Das ist die Uhrzeit, wie sie auf einem Computer angezeigt wird. Ändern Sie die Uhrzeit auf Ihrem Computer, gibt `boost::chrono::system_clock` die neue Uhrzeit an. Wenn Sie Beispiel 37.1 ausführen, wird beispielsweise `13919594042183544 [1/10000000]seconds since Jan 1, 1970` ausgegeben.

Die Epoche ist für `boost::chrono::system_clock` nicht standardisiert. Dass es sich wie an der Ausgabe zu erkennen um den 1. Januar 1970 handelt, ist ein Detail der aktuellen Implementation. Möchten Sie unabhängig von der Implementation die Zeit seit 1. Januar 1970 erhalten, verwenden Sie die Methode `to_time_t()`. Es handelt sich dabei um eine statische Methode, der Sie die aktuelle Systemzeit übergeben und die die Anzahl der Sekunden seit 1. Januar 1970 als std::time_t zurückgibt.

- Bei `boost::chrono::steady_clock` handelt es sich um eine Uhr, die garantiert, dass sie bei einem Zugriff zu einem späteren Zeitpunkt auch eine spätere Uhrzeit zurückgibt. Auch dann, wenn die Uhrzeit auf einem Computer zurückgestellt wird, gibt `boost::chrono::steady_clock` eine spätere Zeit zurück. Auf Englisch wird diese Zeit *monotonic time* genannt.

  Beispiel 37.1 gibt zum Beispiel `10594369282958 nanoseconds since boot` aus. Die Zeit wird abhängig vom letztmaligen Bootvorgang gemessen. Dabei handelt es sich ebenfalls um ein Detail der Implementation, das sich jederzeit ändern kann.

  `boost::chrono::steady_clock` kann nicht auf allen Plattformen angeboten werden. Nur wenn das Makro `BOOST_CHRONO_HAS_CLOCK_STEADY` definiert ist, steht diese Uhr zur Verfügung.

- `boost::chrono::high_resolution_clock` ist eine Typdefinition für `boost::chrono::system_clock` oder `boost::chrono::steady_clock` – je nachdem, welche dieser beiden Uhren Zeit präziser misst. Die Ausgabe ist demnach identisch zu der Uhr, die `boost::chrono::high_resolution_clock` hinterlegt ist.

- `boost::chrono::process_real_cpu_clock` gibt die Zeit an, die ein Programm läuft. Es wird die Zeit seit dem Programmstart gemessen. Wenn Sie Beispiel 37.1 ausführen, wird zum Beispiel `1000000 nanoseconds since process start-up` auf die Standardausgabe ausgegeben.

  Sie können anstatt mit `boost::chrono::process_real_cpu_clock` auch mit `std::clock()` aus `ctime` die Zeit erhalten, die ein Programm läuft. Die aktuelle Implementation von `boost::chrono::process_real_cpu_clock` basiert auf `std::clock()`.

  Beachten Sie, dass Sie `boost::chrono::process_real_cpu_clock` sowie andere Uhren für die CPU-Zeit nur verwenden können, wenn das Makro `BOOST_CHRONO_HAS_PROCESS_CLOCKS` definiert ist.

- `boost::chrono::process_user_cpu_clock` gibt die Zeit an, die ein Programm im *User Space* läuft. Der User Space bezieht sich dabei auf Code eines Programms, der unabhängig vom Betriebssystem läuft. Die Zeit, die benötigt wird, um Code von Betriebssystemfunktionen auszuführen, die in einem Programm aufgerufen werden, wird nicht miteinberechnet.

  `boost::chrono::process_user_cpu_clock` gibt ausschließlich die Zeit an, in der Programmcode im User Space ausgeführt wird. Wird ein Programm zum Beispiel mit einer Funktion wie `Sleep()`, wie sie unter Windows zur Verfügung steht, für einige Zeit angehalten, wird diese Zeit von `boost::chrono::process_user_cpu_clock` nicht miteinberechnet, da während des Aufrufs von `Sleep()` kein Programmcode im User Space ausgeführt wird.

  Beispiel 37.1 gibt zum Beispiel `15600100 nanoseconds since process start-up` aus.

- `boost::chrono::process_system_cpu_clock` ist vergleichbar mit `boost::chrono::process_user_cpu_clock`. Es wird jedoch nicht die Zeit gemessen, die ein Programm im User Space läuft, sondern die es im *Kernel Space* verbringt. `boost::chrono::process_system_cpu_clock` gibt die Zeit an, die für die Ausführung von Betriebssystemfunktionen, die in einem Programm aufgerufen werden, verstrichen ist.

  Beispiel 37.1 kann zum Beispiel `0 nanoseconds since process start-up` ausgeben. Da das Beispielprogramm keine Betriebssystemfunktionen direkt aufruft und auch in Boost.Chrono nur wenige Betriebssystemfunktionen verwendet werden, wird so wenig Code im Kernel Space ausgeführt, dass `boost::chrono::process_system_cpu_clock` 0 zurückgeben kann.

- `boost::chrono::process_cpu_clock` gibt ein Tupel zurück, das die Zeiten enthält, die jeweils einzeln mit `boost::chrono::process_real_cpu_clock`, `boost::chrono::process_user_cpu_clock` und `boost::chrono::process_system_cpu_clock` ermittelt werden können. Beispiel 37.1 gibt beispielsweise `{1000000;15600100;0} nanoseconds since process start-up` aus.

- `boost::chrono::thread_clock` gibt die Zeit an, in der Code in einem Thread ausgeführt wurde. Die von `boost::chrono::thread_clock` gemessene Zeit ist vergleichbar mit der CPU-Zeit, nur dass sie

nicht für ein Programm, sondern für einen Thread gilt. Außerdem wird nicht zwischen User und Kernel Space unterschieden.

`boost::chrono::thread_clock` kann nicht auf allen Plattformen zur Verfügung gestellt werden. Diese Uhr kann nur verwendet werden, wenn das Makro `BOOST_CHRONO_HAS_THREAD_CLOCK` definiert ist.

Boost.Chrono stellt ein weiteres Makro `BOOST_CHRONO_THREAD_CLOCK_IS_STEADY` zur Verfügung, um herauszufinden, ob `boost::chrono::thread_clock` so wie `boost::chrono::steady_clock` monotonic time misst.

Wenn Sie Beispiel 37.1 ausführen, wird zum Beispiel `15600100 nanoseconds since thread st art-up` ausgegeben.

Beachten Sie, dass alle von Boost.Chrono angebotenen Uhren auf Betriebssystemfunktionen angewiesen sind. Es hängt demnach vom Betriebssystem ab, wie genau und verlässlich die zurückgegebenen Zeiten sind.

Beispiel 37.2 Zeiträume von Boost.Chrono

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << p - nanoseconds{1} << '\n';
  std::cout << p + milliseconds{1} << '\n';
  std::cout << p + seconds{1} << '\n';
  std::cout << p + minutes{1} << '\n';
  std::cout << p + hours{1} << '\n';
}
```

Der Aufruf von `now()` gibt für alle Uhren ein Objekt vom Typ `boost::chrono::time_point` zurück. `boost::chrono::time_point` ist eng mit einer Uhr verzahnt, da ein Zeitpunkt nur abhängig von einem absoluten Referenzpunkt gemessen werden kann – und dieser ist in Boost.Chrono per Uhr definiert. `boost::chrono::time_point` ist entsprechend ein Template, das unter anderem eine Uhrenklasse als Parameter erwartet. Uhrenklassen bietet ihrerseits eine Typedefinition für ihren spezialisierten `boost::chrono::time_point` an. process_real_cpu_clock::time_point ist die entsprechende Typdefinition für die Uhrenklasse `process_real_cpu_clock`.
Neben `boost::chrono::time_point` bietet Boost.Chrono eine Klasse `boost::chrono::duration` an, die Zeiträume beschreibt. Da es sich bei `boost::chrono::duration` ebenfalls um ein Template handelt, bietet Boost.Chrono mit `boost::chrono::nanoseconds`, `boost::chrono::milliseconds`, `boost::chrono::microseconds`, `boost::chrono::seconds`, `boost::chrono::minutes` und `boost::chrono::hours` sechs einfach zu verwendende Klassen an.
Boost.Chrono überlädt viele Operatoren, um mit Zeitpunkten und -räumen arbeiten zu können. So werden im Beispiel 37.2 Zeiträume von **p** subtrahiert oder zu **p** hinzuaddiert, um neue Zeitpunkte zu erhalten. Diese werden direkt auf die Standardausgabe ausgegeben.
Wenn Sie Beispiel 37.2 ausführen, stellen Sie fest, dass die Ausgabe aller Zeitpunkte mit der Einheit Nanosekunden erfolgt. Beim Arbeiten mit Zeitpunkten und -räumen verwendet Boost.Chrono automatisch die kleinste Einheit, um sicherzustellen, dass Ergebnisse exakt sind. Möchten Sie einen Zeitpunkt mit einer anderen Einheit verwenden, müssen Sie ihn casten.

Beispiel 37.3 Zeitpunkte casten mit `boost::chrono::time_point_cast()`

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << time_point_cast<minutes>(p) << '\n';
}
```

Boost.Chrono stellt mit `boost::chrono::time_point_cast()` eine Funktion zur Verfügung, die wie ein Cast-Operator verwendet wird. So können Sie wie im Beispiel 37.3 einen Zeitpunkt, der auf der Einheit Nanosekunden basiert, in einen Zeitpunkt umwandeln, der auf Minuten basiert. Da nicht jeder beliebige Zeitpunkt in Nanosekunden gemessen als Zeitpunkt in Minuten dargestellt werden kann, muss `boost::chrono::time_point_cast()` verwendet werden. Die entgegengesetzte Umwandlung von Minuten in Nanosekunden ist implizit möglich – Sie müssen in diesem Fall nicht auf `boost::chrono::time_point_cast()` zugreifen.
Boost.Chrono stellt auch für Zeiträume einen Cast-Operator zur Verfügung, der aus dem gleichen Grund wie `boost::chrono::time_point_cast()` verwendet werden muss.

Beispiel 37.4 Zeiträume casten mit `boost::chrono::duration_cast()`

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  minutes m{1};
  seconds s{35};

  std::cout << m + s << '\n';
  std::cout << duration_cast<minutes>(m + s) << '\n';
}
```

Im Beispiel 37.4 kommt die Funktion `boost::chrono::duration_cast()` zum Einsatz, um einen Zeitraum in Sekunden in einen Zeitraum in Minuten umzuwandeln. Das Beispielprogramm gibt entsprechend `1 minute` aus.

Beispiel 37.5 Zeitpunkte runden

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << floor<minutes>(minutes{1} + seconds{45}) << '\n';
  std::cout << round<minutes>(minutes{1} + seconds{15}) << '\n';
  std::cout << ceil<minutes>(minutes{1} + seconds{15}) << '\n';
}
```

Boost.Chrono bietet neben `boost::chrono::duration_cast()` Funktionen an, um Zeiträume beim Casten zu runden. `boost::chrono::round()` rundet automatisch auf oder ab. `boost::chrono::floor()` rundet immer ab, `boost::chrono::ceil()` immer auf. `boost::chrono::floor()` verwendet `boost::chrono::duration_cast()` – es gibt keinen Unterschied zwischen diesen beiden Funktionen.
Beispiel 37.5 gibt `1 minute`, `1 minute` und `2 minutes` auf die Standardausgabe aus.

Beispiel 37.6 Stream-Manipulatoren für benutzerdefinierte Ausgabe

```
#define BOOST_CHRONO_VERSION 2
#include <boost/chrono.hpp>
#include <boost/chrono/chrono_io.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << symbol_format << minutes{10} << '\n';

  std::cout << time_fmt(boost::chrono::timezone::local, "%H:%M:%S") <<
    system_clock::now() << '\n';
}
```

Boost.Chrono bietet verschiedene Stream-Manipulatoren, um die Ausgabe von Zeiträumen und -punkten zu formatieren. So kann der Manipulator `boost::chrono::symbol_format()` verwendet werden, um die Zeiteinheit mit einem Symbol anstatt mit einem ausgeschriebenen Namen anzugeben. Im Beispiel 37.6 wird entsprechend `10 min` ausgegeben.
Der Manipulator `boost::chrono::time_fmt()` kann verwendet werden, um sowohl eine Zeitzone als auch eine Formatierung festzulegen. Die Zeitzone muss entweder `boost::chrono::timezone::local` oder `boost::chrono::timezone::utc` sein. Die Formatierung ist ein String, in dem mit verschiedenen Kürzeln auf die einzelnen Bestandteile eines Zeitpunkts Bezug genommen wird. So wird im Beispiel 37.6 zum Beispiel `15:46:44` ausgegeben.
Neben Stream-Manipulatoren bietet Boost.Chrono auch zahlreiche Facets. So können Sie mit ihrer Hilfe zum Beispiel Zeiteinheiten in einer anderen Sprache ausgeben.

> **Anmerkung**
>
> Beachten Sie, dass die Input/Output-Funktionen seit Boost 1.52.0 in zwei Versionen vorliegen. Seit Boost 1.55.0 wird standardmäßig die neue Version verwendet. Verwenden Sie eine ältere Version als 1.55.0, müssen Sie wie im Beispiel geschehen das Makro `BOOST_CHRONO_VERSION` definieren und auf 2 setzen.

Kapitel 38

Boost.Timer

Boost.Timer stellt Uhren zur Verfügung, um die Ausführungsgeschwindigkeit von Code zu messen. Auf den ersten Blick konkurriert die Bibliothek mit Boost.Chrono. Während Boost.Chrono jedoch Uhren zur beliebigen Zeitmessung zur Verfügung stellt, geht es bei Boost.Timer explizit um die Ausführungsgeschwindigkeit von Code. Dazu verwendet Boost.Timer unter der Haube Uhren von Boost.Chrono. Wann immer Sie die Performance Ihres Codes messen möchten, sollten Sie sich Boost.Timer zuwenden und nicht Boost.Chrono.
Boost.Timer liegt seit den Boost-Bibliotheken 1.48.0 in einer neuen zweiten Version vor. In Boost.Timer 2 gibt es lediglich eine Headerdatei: Sie binden immer `boost/timer/timer.hpp` ein. Es existiert auch eine Headerdatei `boost/timer.hpp`, die Sie jedoch nicht einbinden dürfen. Diese Headerdatei gehört zur ersten Version von Boost.Timer, die nicht mehr verwendet werden sollte.
Die von Boost.Timer angebotenen Uhren sind in den Klassen `boost::timer::cpu_timer` und `boost::timer::auto_cpu_timer` implementiert. `boost::timer::auto_cpu_timer` ist von `boost::timer::cpu_timer` abgeleitet und stoppt im Destruktor automatisch die Zeit, um anschließend eine Meldung auszugeben.
Im Beispiel 38.1 wird Ihnen zuerst die Klasse `boost::timer::cpu_timer` vorgestellt. Dieses wie auch die folgenden Beispielprogramme führen mathematische Berechnungen aus, damit garantiert Zeit vergeht, bevor die Beispielprogramme enden. Andernfalls würde jeweils 0 gemessen und die Uhren von Boost.Timer nur schlecht vorgestellt werden können.

Beispiel 38.1 Zeit messen mit `boost::timer::cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

Wenn Sie die Klasse `boost::timer::cpu_timer` instanziieren, wird automatisch die Zeitmessung gestartet. Sie können jederzeit die Methode `format()` aufrufen, die die bis zu diesem Moment verstrichene Zeit in einem String zurückgibt. Wenn Sie Beispiel 38.1 ausführen, wird beispielsweise `0.099170s wall, 0.093601s user + 0.000000s system =0.093601s CPU (94.4%)` auf die Standardausgabe ausgegeben.
Boost.Timer misst sowohl die tatsächliche Zeit als auch die CPU-Zeit. Die tatsächliche Zeit gibt an, wie viel Zeit in Wirklichkeit verstrichen ist. Sie könnten diese Zeit mit einer Stoppuhr messen. Die CPU-Zeit gibt den Anteil der tatsächlichen Zeit an, in der Code des Programms ausgeführt wurde. So steht zum einen einem Programm auf heutigen Multitasking-Systemen nicht ununterbrochen ein Prozessor zur Verfügung. Zum anderen kann ein Programm auch absichtlich anhalten, weil zum Beispiel auf eine Benutzereingabe gewartet wird. In diesen Fällen läuft die tatsächliche Zeit weiter, nicht aber die CPU-Zeit.
Boost.Timer unterscheidet bei der CPU-Zeit außerdem zwischen der Zeit, in der Code im *User Space* und im *Kernel Space* ausgeführt wird. Zum Kernel Space gehört Code, der Teil des Betriebssystems ist. Der User Space

ist entsprechend Code, der nicht zum Betriebssystem gehört. Dazu zählt der von Ihnen geschriebene Programmcode als auch Code in Bibliotheken Dritter. So befinden sich zum Beispiel die Boost-Bibliotheken ebenfalls im User Space. Je mehr Funktionen des Betriebssystems Sie aufrufen und je mehr Zeit diese Funktionen benötigen, umso größer wird der Anteil der Zeit im Kernel Space an der CPU-Zeit.

Beispiel 38.2 Zeitmessung unterbrechen und fortsetzen

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.stop();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.resume();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

`boost::timer::cpu_timer` bietet mit `stop()` und `resume()` zwei Methoden an, um die Zeitmessung zu unterbrechen und fortzusetzen. Das führt im Beispiel 38.2 dazu, dass die Zeit, die die zweite `for`-Schleife zur Ausführung benötigt, nicht gemessen wird. Das ist gleichbedeutend mit einer Stoppuhr, die angehalten wird und nach einiger Zeit weiterlaufen darf. So wird beim zweiten Aufruf von `format()` im Beispiel 38.2 eine tatsächliche Zeit und eine CPU-Zeit ausgegeben, als würde die zweite `for`-Schleife im Programm nicht existieren. `boost::timer::cpu_timer` bietet auch eine Methode `start()`. Wenn Sie anstelle von `resume()` `start()` aufrufen, beginnt die Zeitmessung wieder bei null. Der Konstruktor von `boost::timer::cpu_timer` ruft `start()` auf, weswegen die Zeitmessung sofort beginnt, wenn Sie `boost::timer::cpu_timer` instanziieren.

Beispiel 38.3 Tatsächliche Zeit und CPU-Zeit als Tuple erhalten

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);

  cpu_times times = timer.elapsed();
  std::cout << times.wall << '\n';
  std::cout << times.user << '\n';
  std::cout << times.system << '\n';
}
```

Während `format()` die gemessene tatsächliche Zeit und die CPU-Zeit als String zurückgibt, können Sie die

Zeiten auch in einem Tuple erhalten. `boost::timer::cpu_timer` bietet dazu die Methode `elapsed()` an. `elapsed()` gibt ein Tuple vom Typ `boost::timer::times` zurück. Wie im Beispiel 38.3 zu sehen, besitzt dieses Tuple drei Eigenschaften **`wall`**, **`user`** und **`system`**. Die Eigenschaften geben die tatsächliche Zeit und die CPU-Zeit in Nanosekunden an. Es handelt sich um Zahlen vom Typ boost::int_least64_t.
`boost::timer::times` bietet auch eine Methode `clear()` an, um die Eigenschaften **`wall`**, **`user`** und **`system`** auf 0 zu setzen.

Beispiel 38.4 Automatisch messen mit `boost::timer::auto_cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <cmath>

using namespace boost::timer;

int main()
{
  auto_cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
}
```

Sie können die tatsächliche Zeit und die CPU-Zeit eines Code-Blocks mit `boost::timer::auto_cpu_timer` messen. Weil diese Klasse im Destruktor automatisch die Zeit stoppt und eine Meldung auf die Standardausgabe ausgibt, macht Beispiel 38.4 das Gleiche wie Beispiel 38.1.
Die Klasse `boost::timer::auto_cpu_timer` bietet mehrere Konstruktoren an, um zum Beispiel angeben zu können, auf welchen Stream die Datenausgabe erfolgen soll. Standardmäßig ist dies **`std::cout`**.
Sie können sowohl für die Klasse `boost::timer::cpu_timer` als auch für `boost::timer::auto_cpu_timer` angeben, wie Meldungen formatiert werden sollen. Boost.Timer bietet einige wenige Sonderzeichen zur Formatierung an, die ähnlich wie Sonderzeichen von Boost.Format oder `std::printf()` funktionieren. Die Dokumentation von Boost.Timer enthält eine entsprechende Übersicht.

Teil IX

Funktionale Programmierung

In der funktionalen Programmierung sind Funktionen Objekte, die so wie andere Objekte zum Beispiel als Parameter an Funktionen übergeben oder in Containern gespeichert werden können. Es existieren zahlreiche Boost-Bibliotheken, die die funktionale Programmierung in C++ unterstützen.

- Boost.Phoenix ist die umfangreichste und heute wichtigste Boost-Bibliothek. Sie ersetzt die Bibliothek Boost.Lambda, die nur der Vollständigkeit halber und in aller Kürze vorgestellt wird.
- Boost.Function stellt eine Klasse zur Verfügung, die es einfacher macht, Funktionszeiger zu definieren, ohne die aus der Programmiersprache C stammende Syntax verwenden zu müssen.
- Boost.Bind ist ein Adapter, der es möglich macht, Funktionen als Parameter an andere Funktionen zu übergeben, wenn sich die tatsächliche und die erwartete Funktionssignatur unterscheiden.
- Boost.Ref kann im Zusammenhang mit Funktionen verwendet werden, um eine Referenz auf ein Objekt zu übergeben, auch wenn ein Funktion eine Kopie erwartet.
- Boost.Lambda kann als Vorgänger von Boost.Phoenix bezeichnet werden. So handelt es sich um eine alte Bibliothek, die es möglich machte, Lambda-Funktionen zu verwenden, bevor diese Jahre später mit C++11 Teil der Programmiersprache wurden.

Kapitel 39

Boost.Phoenix

Boost.Phoenix ist die wichtigste Boost-Bibliothek für die funktionale Programmierung. Während Bibliotheken wie Boost.Bind oder Boost.Lambda ebenfalls die funktionale Programmierung unterstützen, schließt der Funktionsumfang von Boost.Phoenix diese Bibliotheken ein und geht darüber hinaus.
In der funktionalen Programmierung sind Funktionen Objekte und können wie Objekte verarbeitet werden. So ist es mit Boost.Phoenix möglich, eine Funktion als Ergebnis einer anderen Funktion zurückzugeben. Es ist auch möglich, eine Funktion als Parameter an eine andere Funktion zu übergeben. Da Funktionen Objekte sind, kann zwischen Instanziierung und Ausführung unterschieden werden. Ein Zugriff auf eine Funktion ist nicht gleichbedeutend mit einem Aufruf.
Boost.Phoenix unterstützt die funktionale Programmierung mit Hilfe von Funktionsobjekten: Funktionen sind Objekte, die auf Klassen basieren, die den Operator `operator()` überladen. So verhalten sich Funktionsobjekte tatsächlich wie andere Objekte in C++. Sie können kopiert und zum Beispiel in einem Container gespeichert werden. Sie verhalten sich aber auch wie Funktionen, da sie aufgerufen werden können.
Die funktionale Programmierung ist in C++ nicht neu. Sie können auch ohne Boost.Phoenix eine Funktion als Parameter an eine andere Funktion übergeben.

Beispiel 39.1 Prädikate als globable Funktion, Lambda-Funktion und Phoenix-Funktion

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  std::cout << std::count_if(v.begin(), v.end(), is_odd) << '\n';

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Beispiel 39.1 greift auf den Algorithmus `std::count_if()` zu, um ungerade Zahlen im Vektor **`v`** zu zählen. `std::count_if()` wird dreimal aufgerufen: Einmal mit einem Prädikat in Form einer freistehenden Funktion, einmal mit einer Lambda-Funktion und einmal mit einer Phoenix-Funktion.
Die Phoenix-Funktion unterscheidet sich von der freistehenden Funktion und der Lambda-Funktion darin, dass sie kein umschließendes Gerüst besitzt. Während in den anderen beiden Fällen ein Funktionskopf mit Parameterliste vorhanden ist, besteht die Phoenix-Funktion quasi ausschließlich aus einer Implementation.
Der entscheidende Bestandteil der Phoenix-Funktion ist **`boost::phoenix::placeholders::arg1`**. Es handelt sich hierbei um eine Instanz eines Funktionsobjekts. Sie können auf **`arg1`** ähnlich wie auf **`std::cout`** zugreifen: Die Objekte existieren, wenn Sie die entsprechende Headerdatei einbinden.

Mit **arg1** definieren Sie eine unäre Funktion. Der Ausdruck `arg1 % 2 ==1` führt dazu, dass eine neue Funktion entsteht, die genau einen Parameter erwartet. Die Funktion wird nicht sofort ausgeführt, sondern in **phoenix** gespeichert. **phoenix** wird an `std::count_if()` übergeben, von wo die Funktion für jede Zahl in **v** aufgerufen wird.
arg1 ist ein Platzhalter für den Wert, der beim Aufruf der Phoenix-Funktion übergeben wird. Da hier ausschließlich **arg1** verwendet wird, handelt es sich um eine unäre Funktion. Boost.Phoenix bietet weitere Platzhalter wie **boost::phoenix::placeholders::arg2** und **boost::phoenix::placeholders::arg3** an. Eine Phoenix-Funkion hat immer genauso viele Parameter wie der Platzhalter mit der höchsten Zahl.
Wenn Sie Beispiel 39.1 ausführen, wird dreimal `3` ausgegeben.

Beispiel 39.2 Phoenix-Funktion vs. Lambda-Funktion

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  std::vector<long> v2;
  v2.insert(v2.begin(), v.begin(), v.end());

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
  std::cout << std::count_if(v2.begin(), v2.end(), phoenix) << '\n';
}
```

Beispiel 39.2 hebt einen entscheidenden Unterschied zwischen Phoenix- und Lambda-Funktionen hervor. Phoenix-Funktionen benötigen nicht nur keinen Funktionskopf mit Parameterliste. Ihre Parameter sind außerdem typenlos. So erwartet die Lambda-Funktion **lambda** einen Parameter vom Typ int. Die Phoenix-Funktion **phoenix** akzeptiert hingegen jeden beliebigen Typ, solange auf ihn der Modulo-Operator mit einer Zahl angewandt werden kann.
Sie können sich Phoenix-Funktionen als Template-Funktionen vorstellen. So wie Template-Funktionen beliebige Typen akzeptieren, ist eine Phoenix-Funktion ebenfalls nicht an bestimmte Typen gebunden. Das ist der Grund, warum **phoenix** im Beispiel 39.2 als Prädikat sowohl für den Container **v** als auch **v2** verwendet werden kann, obwohl die Zahlen in diesen Containern unterschiedliche Typen haben. Würden Sie versuchen, **lambda** mit **v2** zu verwenden, gäbe es einen Compiler-Fehler.

> Anmerkung
>
> Seit C++14 ist es möglich, generische Lambda-Funktionen zu definieren, die Phoenix-Funktionen in nichts nachstehen. Wird der Parameter **i** in der Lambda-Funktion im obigen Beispiel mit `auto` anstelle des Typs `int` definiert, kann die Lambda-Funktion auf beide Container **v** und **v2** angewandt werden.

Beispiel 39.3 Phoenix-Funktionen als verspätet ausgeführter C++-Code

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};
```

```
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > 2 && arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Im Beispiel 39.3 sehen Sie eine Phoenix-Funktion, die als Prädikat für `std::count_if()` alle ungeraden Zahlen größer als 2 zählt. Dazu wird in der Phoenix-Funktion zweimal auf **`arg1`** zugegriffen: Einmal, um den Platzhalter auf größer als 2 zu vergleichen, einmal, um auf ungerade zu überprüfen. Die beiden Bedingungen werden mit einem `&&` verknüpft.
Sie können sich Phoenix-Funktionen als C++-Code vorstellen, der nicht sofort, sondern verspätet ausgeführt wird. Die Phoenix-Funktion im Beispiel 39.3 sieht wie eine herkömmliche Bedingung aus, in der mehrere logische und arithmetische Operatoren verwendet werden. Die Bedingung wird jedoch nicht sofort ausgeführt, sondern erst dann, wenn auf sie innerhalb von `std::count_if()` zugegriffen wird. Bei diesem Zugriff handelt es sich um einen herkömmlichen Funktionsaufruf.
Beispiel 39.3 gibt `2` aus.

Beispiel 39.4 Explizite Phoenix-Typen

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > val(2) && arg1 % val(2) == val(1);
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Beispiel 39.4 verwendet explizite Typen für alle Operanden in der Phoenix-Funktion. Genaugenommen sehen Sie keine Typen, sondern lediglich die Hilfsfunktion `boost::phoenix::val()`. Diese Funktion gibt ein Funktionsobjekt zurück, das mit dem Wert initialisiert wird, der an die Funktion übergeben wird. Der konkrete Typ des Funktionsobjekts spielt keine Rolle. Entscheidend ist, dass Boost.Phoenix für verschiedene Typen Operatoren wie `>`, `&&`, `%` und `==` überlädt. Somit werden Bedingungen nicht sofort überprüft. Stattdessen werden Funktionsobjekte verknüpft, um mächtigere Funktionsobjekte zu erstellen. Je nach Zusammensetzung der Operanden werden diese automatisch als Funktionsobjekte erkannt. Falls nicht, können Sie explizit auf Hilfsfunktionen wie `val()` zugreifen.

Beispiel 39.5 `boost::phoenix::placeholders::arg1` und `boost::phoenix::val()`

```
#include <boost/phoenix/phoenix.hpp>
#include <iostream>

int main()
{
  using namespace boost::phoenix::placeholders;
  std::cout << arg1(1, 2, 3, 4, 5) << '\n';

  auto v = boost::phoenix::val(2);
  std::cout << v() << '\n';
}
```

Beispiel 39.5 verdeutlicht, wie **`arg1`** und `val()` funktionieren. **`arg1`** ist eine Instanz eines Funktionsobjekts. Sie kann direkt verwendet und wie eine Funktion aufgerufen werden. Sie können beliebig viele Parameter übergeben – **`arg1`** gibt den ersten zurück.
`val()` ist eine Funktion, um eine Instanz eines Funktionsobjekts zu erstellen. Das entsprechende Funktionsobjekt wird mit einem Wert initialisiert. Wird auf die Instanz wie auf eine Funktion zugegriffen, wird der Wert zurückgegeben.
Wenn Sie Beispiel 39.5 ausführen, wird `1` und `2` ausgegeben.

Beispiel 39.6 Eigene Phoenix-Funktionen erstellen

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

struct is_odd_impl
{
    typedef bool result_type;

    template <typename T>
    bool operator()(T t) const { return t % 2 == 1; }
};

boost::phoenix::function<is_odd_impl> is_odd;

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

Im Beispiel 39.6 sehen Sie, wie Sie eine eigene Phoenix-Funktion erstellen. Sie verwenden dazu das Template `boost::phoenix::function`, dem Sie ein Funktionsobjekt übergeben. Im Beispiel handelt es sich um die Klasse `is_odd_impl`. Diese Klasse überlädt den Operator `operator()` derart, dass für eine ungerade Zahl, die als Parameter übergeben wird, `true` und für eine gerade Zahl `false` zurückgegeben wird.
Beachten Sie, dass Sie den Typ result_type definieren müssen. Boost.Phoenix greift auf ihn zu, um den Typ des Rückgabewerts des Operators `operator()` zu ermitteln.
`is_odd()` ist eine Funktion, die Sie genauso verwenden können wie `val()`. Beide Funktionen geben ein Funktionsobjekt zurück. Beim Aufruf des Funktionsobjekts werden Parameter, die an die Funktionen übergeben werden, an den Operator `operator()` weitergereicht. Für Beispiel 39.6 bedeutet dies, dass `std::count_if()` immer noch ungerade Zahlen zählt.

Beispiel 39.7 Freistehende Funktionen in Phoenix-Funktionen umwandeln

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool is_odd_function(int i) { return i % 2 == 1; }

BOOST_PHOENIX_ADAPT_FUNCTION(bool, is_odd, is_odd_function, 1)

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

Wenn Sie eine existierende freistehende Funktion in eine Phoenix-Funktion umwandeln möchten, können Sie dies wie im Beispiel 39.7 tun. Sie müssen nicht unbedingt wie im vorherigen Beispiel ein Funktionsobjekt definieren.
Sie verwenden das Makro `BOOST_PHOENIX_ADAPT_FUNCTION`, um aus einer freistehenden Funktion eine Phoenix-Funktion zu machen. Übergeben Sie dem Makro zuerst den Typ des Rückgabewerts, dann den Namen der zu definierenden Boost.Phoenix-Funktion, dann den Namen der freistehenden Funktion und zuletzt die Zahl der Parameter, die die freistehende Funktion erwartet.

Beispiel 39.8 Phoenix-Funktionen mit `boost::phoenix::bind()`

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), bind(is_odd, arg1)) << '\n';
}
```

Möchten Sie eine freistehende Funktion als Phoenix-Funktion verwenden, können Sie auch wie im Beispiel 39.8 auf `boost::phoenix::bind()` zugreifen. `boost::phoenix::bind()` funktioniert genauso wie `std::bind()`. Sie übergeben als ersten Parameter den Namen der freistehenden Funktion. Alle weiteren Parameter werden an die freistehende Funktion beim Aufruf weitergereicht.

> **Tipp**
>
> Vermeiden Sie `boost::phoenix::bind()`. Erstellen Sie Ihre eigenen Phoenix-Funktionen. Dies führt zu lesbarerem Code. Gerade bei komplexen Ausdrücken mit mehreren Verknüpfungen ist es wenig hilfreich, sich zusätzlich mit den Details von `boost::phoenix::bind()` beschäftigen zu müssen.

Beispiel 39.9 Beliebig komplexe Phoenix-Funktionen

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  int count = 0;
  std::for_each(v.begin(), v.end(), if_(arg1 > 2 && arg1 % 2 == 1)
    [
      ++ref(count)
    ]);
  std::cout << count << '\n';
}
```

Da Funktionsobjekte beliebig implementiert sein können, bietet Boost.Phoenix einige an, die Schlüsselwörter aus C++ simulieren. So können Sie wie im Beispiel 39.8 die Funktion `boost::phoenix::if_()` aufrufen, um ein Funktionsobjekt zu erstellen, das sich wie `if` verhält und eine Bedingung überprüft. Ist die Bedingung wahr, wird Code ausgeführt, der mit `operator[]` an das Funktionsobjekt übergeben wurde. Dabei muss dieser Code natürlich wieder in Form von Funktionsobjekten vorliegen. So können beliebig komplexe Phoenix-Funktionen erstellt werden.
Im Beispiel 39.8 wird die Variable **`count`** für jede ungerade Zahl größer als 2 inkrementiert. Damit der Inkrement-Operator nicht direkt auf **`count`** angewandt wird, wird **`count`** mit Hilfe der Funktion `boost::phoenix::ref()`

in ein Funktionsobjekt gepackt. Im Gegensatz zu `boost::phoenix::val()` wird kein Wert in das Funktionsobjekt kopiert. Das von `boost::phoenix::ref()` zurückgegebene Funktionsobjekt speichert eine Referenz – hier eine Referenz auf **`count`**.

> Tipp
>
> Verwenden Sie Boost.Phoenix nicht, um komplexe Funktionen zu erstellen. Greifen Sie in diesem Fall auf C++11-Lambda-Funktionen zu. Während Boost.Phoenix der Syntax von C++ sehr nahe kommt, tragen vermeintliche Schlüsselwörter wie `if_` oder Code-Blöcke, die aus eckigen Klammern bestehen, nicht unbedingt zur besseren Lesbarkeit bei.

Kapitel 40

Boost.Function

Boost.Function stellt die Klasse `boost::function` zur Verfügung, um Funktionszeiger zu kapseln. Die Klasse ist in der Headerdatei `boost/function.hpp` definiert.
Wenn Sie in einer Entwicklungsumgebung arbeiten, die C++11 unterstützt, haben Sie Zugriff auf eine Klasse `std::function`. Sie finden sie in der Headerdatei `functional`. In diesem Fall können Sie Boost.Function ignorieren, da `boost::function` und `std::function` gleich sind.

Beispiel 40.1 `boost::function` in Aktion

```
#include <boost/function.hpp>
#include <iostream>
#include <cstdlib>
#include <cstring>

int main()
{
  boost::function<int(const char*)> f = std::atoi;
  std::cout << f("42") << '\n';
  f = std::strlen;
  std::cout << f("42") << '\n';
}
```

Mit `boost::function` wird ein Zeiger auf eine Funktion mit einer bestimmten Signatur definiert. Im Beispiel 40.1 ist angegeben, dass **`f`** auf Funktionen zeigen darf, deren Rückgabewert vom Typ int ist und die einen einzigen Parameter vom Typ const char* erwarten. Einmal definiert können Funktionen an **`f`** gebunden werden, deren Signatur kompatibel ist. So werden im Beispiel 40.1 die Funktionen `std::atoi()` und `std::strlen()` an **`f`** gebunden.
Beachten Sie, dass keine perfekte Übereinstimmung der Typen notwendig ist: Auch wenn `std::strlen()` so definiert ist, dass diese Funktion einen Rückgabewert vom Typ std::size_t besitzt, kann sie trotzdem an **`f`** gebunden werden.
Da **`f`** ein Funktionszeiger ist, kann die an **`f`** gebundene Funktion über den überladenen Operator `operator()` aufgerufen werden. Je nachdem, an welche Funktion **`f`** gebunden ist, wird im Beispiel 40.1 `std::atoi()` oder `std::strlen()` aufgerufen.
Wenn **`f`** an keine Funktion gebunden ist, wird bei einem Aufruf die Ausnahme `boost::bad_function_call` geworfen. Beispiel 40.2 demonstriert dies.

Beispiel 40.2 `boost::bad_function_call` bei Aufruf ohne Funktionsbindung

```
#include <boost/function.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::function<int(const char*)> f;
    f("");
  }
  catch (boost::bad_function_call &ex)
  {
```

```
    std::cerr << ex.what() << '\n';
  }
}
```

Beachten Sie, dass Sie durch die Zuweisung von `nullptr` einen Funktionszeiger vom Typ `boost::function` zurücksetzen und die Bindung an eine Funktion lösen können. Sollten Sie dann versuchen, eine Funktion aufzurufen, wird ebenfalls eine Ausnahme vom Typ `boost::bad_function_call` geworfen. Die von `boost::function` zur Verfügung gestellten Methoden `empty()` und `operator bool` können Sie nutzen, um herauszufinden, ob ein Objekt vom Typ `boost::function` auf eine Funktion verweist.
Boost.Function ermöglicht auch, Methoden an Objekte vom Typ `boost::function` zu binden. Beim Aufruf muss das Objekt angegeben werden, für das die Methode ausgeführt werden soll. Sehen Sie sich dazu Beispiel 40.3 an.

Beispiel 40.3 Eine Methode an `boost::function` binden

```
#include <boost/function.hpp>
#include <functional>
#include <iostream>

struct world
{
  void hello(std::ostream &os)
  {
    os << "Hello, world!\n";
  }
};

int main()
{
  boost::function<void(world*, std::ostream&)> f = &world::hello;
  world w;
  f(&w, std::ref(std::cout));
}
```

Als erster Parameter muss beim Methodenaufruf das Objekt angegeben werden, für das die an **`f`** gebundene Methode ausgeführt werden soll. Das setzt voraus, dass der erste Parameter in runden Klammern bei der Template-Definition ein Zeiger auf die entsprechende Klasse ist. Die nachfolgenden Parameter kennzeichnen die Signatur der entsprechenden Methode.

Kapitel 41

Boost.Bind

Boost.Bind ist eine Bibliothek, die etwas vereinfachen und verallgemeinern soll, was ursprünglich den Einsatz von `std::bind1st()` und `std::bind2nd()` voraussetzte. Diese beiden Funktionen wurden mit C++98 in die Standardbibliothek aufgenommen und erlauben es, Funktionen zu verknüpfen, selbst wenn ihre Signaturen nicht kompatibel sind. Das ist zum Beispiel ein Problem, wenn Funktionen mit Algorithmen aus der Standardbibliothek verwendet werden sollen, sie aber mehr Parameter erwarten als ihnen beim Aufruf vom Algorithmus übergeben werden.
Boost.Bind wurde mit C++11 in die Standardbibliothek aufgenommen. Wenn Ihre Entwicklungsumgebung C++11 unterstützt, finden Sie in der Headerdatei `functional` eine Funktion `std::bind()`. Die Probleme, die Boost.Bind und `std::bind()` lösen, können Sie jedoch je nach Anwendungsfall womöglich besser mit Lambda-Funktionen oder Boost.Phoenix lösen.

Beispiel 41.1 `std::for_each()` mit passender Funktion

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

void print(int i)
{
  std::cout << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), print);
}
```

`std::for_each()` erwartet als dritten Parameter eine Funktion oder ein Funktionsobjekt, die ihrerseits einen einzigen Parameter erwarten. Im Beispiel 41.1 übergibt `std::for_each()` die Zahlen im Container **v** nacheinander jeweils als einzigen Parameter an `print()`.
Soll eine Funktion aufgerufen werden, deren Signatur nicht den Anforderungen eines Algorithmus entspricht, wird es schwierig. Möchten Sie zum Beispiel `print()` derart ändern, dass die Ausgabe nicht mehr standardmäßig auf **`std::cout`** erfolgt, sondern der Stream über einen zweiten Parameter an die Funktion übergeben werden kann, könnte `print()` nicht ohne Weiteres mit `std::for_each()` verwendet werden.

Beispiel 41.2 `std::for_each()` mit `std::bind1st()`

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

class print : public std::binary_function<std::ostream*, int, void>
{
public:
  void operator()(std::ostream *os, int i) const
  {
    *os << i << '\n';
```

```
  }
};

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), std::bind1st(print{}, &std::cout));
}
```

Beispiel 41.2 gibt wie das vorherige alle Zahlen aus **v** auf die Standardausgabe aus. Diesmal ist es jedoch möglich, den Stream, auf den die Ausgabe erfolgen soll, als Parameter zu übergeben. Dies erforderte jedoch größere Änderungen: So musste die Funktion `print()` in ein Funktionsobjekt umgewandelt werden, das außerdem von `std::binary_function` abgeleitet werden muss.
Mit Boost.Bind kann eine einfachere Lösung gewählt werden, ohne `print()` von einer Funktion in ein Funktionsobjekt umwandeln zu müssen. Dazu wird auf die Template-Funktion `boost::bind()` zugegriffen, die in der Headerdatei `boost/bind.hpp` definiert ist.

Beispiel 41.3 `std::for_each()` mit `boost::bind()`

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

void print(std::ostream *os, int i)
{
  *os << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), boost::bind(print, &std::cout, _1));
}
```

Beispiel 41.3 verwendet `print()` als Funktion und nicht als Funktionsobjekt. Da `print()` zwei Parameter erwartet, kann die Funktion nicht direkt an `std::for_each()` übergeben werden. Stattdessen wird `boost::bind()` an `std::for_each()` übergeben. `print()` wiederum wird als erster Parameter an `boost::bind()` übergeben.
Da `print()` zwei Parameter erwartet, müssen zwei zusätzliche Parameter an `boost::bind()` übergeben werden. Das ist zum einen ein Zeiger auf **`std::cout`**, zum anderen die Angabe **`_1`**.
`_1` ist ein Platzhalter, der in Boost.Bind definiert ist. Neben **`_1`** sind weitere Platzhalter von **`_2`** bis **`_9`** definiert. Diese Platzhalter führen dazu, dass `boost::bind()` ein Funktionsobjekt zurückgibt, dem genauso viele Parameter übergeben werden müssen wie der Platzhalter mit der höchsten Zahl. Ist so wie im Beispiel 41.3 lediglich der Platzhalter **`_1`** angegeben, gibt `boost::bind()` ein unäres Funktionsobjekt zurück – also ein Funktionsobjekt, das einen einzigen Parameter erwartet. Dies ist hier insofern notwendig, weil `std::for_each()` genau einen Parameter übergibt.
Wenn Sie das Beispiel ausführen, ruft `std::for_each()` ein unäres Funktionsobjekt auf. Der Wert, der an das unäre Funktionsobjekt übergeben wird – eine Zahl aus dem Container **`v`** – nimmt die Position des Platzhalters **`_1`** ein. Der Zeiger auf **`std::cout`** und die Zahl werden als Parameter an die Funktion `print()` weitergereicht. Auf diese Weise sieht `std::for_each()` ein unäres Funktionsobjekt, das durch die Art und Weise, wie `boost::bind()` verwendet wird, definiert ist. `boost::bind()` ruft seinerseits die Funktion auf, die als erster Paramter angegeben ist, und übergibt den Zeiger auf **`std::cout`** und die Zahl im Platzhalter an diese Funktion.
Beachten Sie, dass `boost::bind()` so wie `std::bind1st()` und `std::bind2nd()` Parameter als Kopie übernimmt. Um in den obigen Beispielen zu verhindern, dass versucht wird, von **`std::cout`** eine Kopie zu erstellen, erwartet `print()` einen Zeiger auf einen Stream. Im Kapitel 42 lernen Sie eine Funktion kennen, die es Ihnen ermöglicht, Parameter als Referenz zu übergeben.
Im Folgenden sehen Sie ein Beispiel, in dem mit Hilfe von `boost::bind()` ein binäres Funktionsobjekt definiert wird. Dazu wird der Algorithmus `std::sort()` verwendet, der als dritten Parameter eine binäre Funktion erwartet.

Beispiel 41.4 `std::sort()` mit `boost::bind()`

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _1, _2));
  for (int i : v)
    std::cout << i << '\n';
}
```

Im Beispiel 41.4 wird ein binäres Funktionsobjekt erstellt, weil mit **_2** ein Platzhalter verwendet wird, der die Zwei im Namen trägt. Der Algorithmus `std::sort()` ruft dieses binäre Funktionsobjekt mit zwei Werten aus dem Container **v** auf und wertet den Rückgabewert aus, um den Container zu sortieren. So, wie die Funktion `compare()` definiert ist, wird **v** absteigend sortiert.
Da `compare()` eine binäre Funktion ist, könnte sie direkt an `std::sort()` übergeben werden. Der Einsatz von `boost::bind()` kann trotzdem Sinn ergeben, wenn zum Beispiel der Container aufsteigend sortiert werden soll, die Definition der Funktion `compare()` aber nicht geändert werden soll.

Beispiel 41.5 `std::sort()` mit `boost::bind()` und geänderter Platzhalterreihenfolge

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _2, _1));
  for (int i : v)
    std::cout << i << '\n';
}
```

Im Beispiel 41.5 ist die Reihenfolge der Platzhalter geändert worden: **_2** wird als erster und **_1** als zweiter Parameter an `compare()` übergeben. Somit wird **v** aufsteigend sortiert.

Kapitel 42

Boost.Ref

Die Bibliothek Boost.Ref definiert zwei Funktionen `boost::ref()` und `boost::cref()` in der Headerdatei `boost/ref.hpp`. Sie sind von Bedeutung, wenn zum Beispiel eine Funktion mit `std::bind()` verknüpft werden soll, die als Parameter eine Referenz erwartet. Weil `std::bind()` Parameter kopiert, müssen Referenzen besonders gehandhabt werden.
Boost.Ref ist mit C++11 in die Standardbibliothek eingegangen. Sie finden dort die Funktionen `std::ref()` und `std::cref()` in der Headerdatei `functional`.

Beispiel 42.1 `boost::ref()` in Aktion

```
#include <boost/ref.hpp>
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

void print(std::ostream &os, int i)
{
  os << i << std::endl;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::bind(print, boost::ref(std::cout), std::placeholders::_1));
}
```

Im Beispiel 42.1 soll die Funktion `print()` als dritter Parameter an `std::for_each()` übergeben werden, um alle Zahlen in **`v`** auf einen Stream auszugeben. Weil `print()` zwei Parameter akzeptiert – einen Stream und eine Zahl, die auf den Stream ausgegeben werden soll – wird `std::bind()` verwendet. Über `std::bind()` wird als erster Parameter an `print()` der Stream **`std::cout`** übergeben. Da `print()` den Stream als Referenz erwartet, `std::bind()` Parameter jedoch als Kopie übernimmt, wird **`std::cout`** mit Hilfe der Funktion `boost::ref()` gekapselt. Diese Funktion gibt ein Proxy-Objekt zurück, das eine Referenz auf das Objekt enthält, das an `boost::ref()` übergeben wurde. Auf diese Weise ist es möglich, eine Referenz auf **`std::cout`** zu übergeben, obwohl `std::bind()` Parameter als Kopie übernimmt.
Mit der Template-Funktion `boost::cref()` kann eine konstante Referenz übergeben werden.

Kapitel 43

Boost.Lambda

Vor C++11 war es notwendig, auf eine Bibliothek wie Boost.Lambda zuzugreifen, um Lambda-Funktionen verwenden zu können. Seit C++11 kann diese Bibliothek als veraltet bezeichnet werden, da Lambda-Funktionen seit dieser Version des Standards Teil der Programmiersprache sind. Arbeiten Sie in einer Entwicklungsumgebung, die C++11 nicht unterstützt, sollten Sie außerdem zuerst einen Blick auf Boost.Phoenix werfen, bevor Sie sich Boost.Lambda zuwenden. Boost.Phoenix ist die neuere Bibliothek, um Lambda-Funktionen ohne C++11 verwenden zu können.
Das Ziel von Lambda-Funktionen ist es, den Code kompakter und damit verständlicher zu machen. Sehen Sie sich dazu Beispiel 43.1 an.

Beispiel 43.1 `std::for_each()` mit einer Boost.Lambda-Funktion

```
#include <boost/lambda/lambda.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::cout << boost::lambda::_1 << "\n");
}
```

Boost.Lambda stellt verschiedene Hilfsmittel zur Verfügung, um namenlose Funktionen zu erstellen. Code wird dort angegeben, wo er ausgeführt werden soll, ohne ihn in eine Funktion packen und diese aufrufen zu müssen. So wird mit `std::cout << boost::lambda::_1 << "\n"` eine Lambda-Funktion definiert, die einen Parameter erwartet und diesen gefolgt von einem Zeilenumbruch in die Standardausgabe schreibt.
Bei **`boost::lambda::_1`** handelt es sich um einen Platzhalter. Mit **`boost::lambda::_2`** und **`boost::lambda::_3`** stellt Boost.Lambda zwei weitere Platzhalter zur Verfügung. Verwenden Sie lediglich **`boost::lambda::_1`**, erstellen Sie eine Lambda-Funktion, die einen Parameter erwartet. Verwenden Sie **`boost::lambda::_2`** – zusätzlich zu **`boost::lambda::_1`** oder als einzigen Platzhalter – erwartet die Lambda-Funktion zwei Parameter. Die Signatur der Lambda-Funktion hängt von der höchsten Ziffer der verwendeten Platzhalter ab. Im Beispiel 43.1 darf in der Lambda-Funktion nur der Platzhalter **`boost::lambda::_1`** verwendet werden, weil die Lambda-Funktion an `std::for_each()` übergeben wird und dieser Algorithmus eine unäre Funktion erwartet.
Beachten Sie, dass Sie die Headerdatei `boost/lambda/lambda.hpp` einbinden müssen, um Platzhalter verwenden zu können.
Beachten Sie außerdem, dass im Beispiel 43.1 „\n“ und nicht `std::endl` für einen Zeilenumbruch verwendet wird. Mit `std::endl` könnte der Code nicht kompiliert werden. Weil der Typ der Lambda-Funktion `std::cout << boost::lambda::_1` ein anderer ist als der, den die unäre Template-Funktion `std::endl()` kennt und erwartet, kann `std::endl` nicht verwendet werden.

Beispiel 43.2 Eine Boost.Lambda-Funktion mit `boost::lambda::if_then()`

```
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/if.hpp>
#include <vector>
#include <algorithm>
```

```
#include <iostream>

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    boost::lambda::if_then(boost::lambda::_1 > 1,
    std::cout << boost::lambda::_1 << "\n"));
}
```

In der Headerdatei `boost/lambda/if.hpp` definiert Boost.Lambda mehrere Konstrukte, um `if`-Kontrollstrukturen in einer Lambda-Funktion zu verwenden. Das einfachste Konstrukt ist die Template-Funktion `boost::lambda::if_then()`, die zwei Parameter erwartet: Im ersten Parameter wird eine Bedingung überprüft. Ist sie wahr, wird der zweite Parameter ausgeführt. Beide Parameter können wie im Beispiel 43.2 Lambda-Funktionen sein. Neben `boost::lambda::if_then()` stellt Boost.Lambda die Template-Funktionen `boost::lambda::if_then_else()` und `boost::lambda::if_then_else_return()` zur Verfügung, die beide drei Parameter erwarten. Darüber hinaus gibt es Template-Funktionen für Schleifen, Cast-Operatoren und für `throw`, um Ausnahmen in Lambda-Funktionen zu werfen. Die zahlreichen von Boost.Lambda definierten Template-Funktionen ermöglichen es, beliebige Lambda-Funktionen zu definieren, die herkömmlichen Funktionen in C++ in nichts nachstehen.

Teil X

Parallele Programmierung

Die folgenden Bibliotheken unterstützen die parallele Programmierung.

- Mit Boost.Thread erstellen und verwalten Sie Ihre eigenen Threads.
- Boost.Atomic verwenden Sie, wenn Sie von mehreren Threads aus auf Variablen integraler Typen über atomare Operationen zugreifen möchten.
- Boost.Lockfree bietet thread-safe Container an.
- Boost.MPI stammt aus dem Supercomputer-Umfeld und unterstützt die parallele Programmierung insofern, als dass Ihr Programm mehrfach gestartet und gleichzeitig in mehreren Prozessen ausgeführt wird. Sie konzentrieren sich auf die Programmierung der parallel auszuführenden Aufgaben, Boost.MPI übernimmt die Koordination der Prozesse. Bei Boost.MPI müssen Sie sich nicht um Details wie die Synchronisation von Zugriffen auf gemeinsam genutzte Daten kümmern. Boost.MPI setzt jedoch eine entsprechende Laufzeitumgebung voraus.

Kapitel 44

Boost.Thread

Boost.Thread ist die Bibliothek, die es Ihnen ermöglicht, Threads zu verwenden. Sie stellt darüberhinaus Klassen zur Verfügung, um den Zugriff auf Daten zu synchronisieren, die von mehreren Threads verwendet werden. Threads werden seit C++11 von der Standardbibliothek unterstützt. So finden Sie in der Standardbibliothek ebenfalls Klassen, mit denen Threads erstellt und Zugriffe auf Daten synchronisiert werden können. Boost.Thread ähnelt in großen Teilen der Standardbibliothek, bietet jedoch zahlreiche Erweiterungen an. So können Threads, die mit Boost.Thread erstellt werden, unterbrochen werden. Sie finden in Boost.Thread außerdem spezielle Locks, die erst mit C++14 in die Standardbibliothek aufgenommen wurden. Es kann daher sinnvoll sein, Boost.Thread einzusetzen, selbst wenn Sie in einer Entwicklungsumgebung arbeiten, die C++11 unterstützt.

44.1 Threads erstellen und verwalten

Die wichtigste Klasse in Boost.Thread ist `boost::thread`. Sie ist in der Headerdatei `boost/thread.hpp` definiert und wird verwendet, um einen Thread zu erstellen. Im Beispiel 44.1 sehen Sie, wie sie angewandt wird.

Beispiel 44.1 `boost::thread` in Aktion

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << '\n';
  }
}

int main()
{
  boost::thread t{thread};
  t.join();
}
```

Dem Konstruktor von `boost::thread` wird der Name der Funktion übergeben, die als Thread gestartet werden soll. Wenn im Beispiel 44.1 die Variable **`t`** erstellt wird, beginnt die Funktion `thread()` sofort, in einem eigenen Thread zu laufen. Die Funktion `thread()` wird ab diesem Zeitpunkt gleichzeitig zur Funktion `main()` ausgeführt.
Weil das Programm wie gewohnt nach Ablauf der Funktion `main()` beendet werden würde, wird für den soeben gestarteten Thread `join()` aufgerufen. Es handelt sich dabei um eine blockierende Methode: `join()` hält den

aktuellen Thread an und kehrt erst dann zurück, wenn der Thread, für den `join()` aufgerufen wurde, beendet wurde. Dies bedeutet für Beispiel 44.1, dass `main()` solange wartet, bis `thread()` seine Arbeit getan hat und zurückkehrt.
Beachten Sie, dass Sie über eine Variable wie **`t`** auf einen Thread zugreifen können, um zum Beispiel mit `join()` auf dessen Beendigung zu warten. Der Thread würde aber auch dann weiterlaufen, wenn der Gültigkeitsbereich von **`t`** enden und die Variable zerstört werden würde. Ein Thread ist anfangs an eine Variable vom Typ `boost::thread` gebunden, läuft aber auch dann weiter, wenn die Variable nicht mehr existiert. Sie können sogar eine Methode `detach()` aufrufen, mit der Sie eine Variable vom Typ `boost::thread` von einem Thread entkoppeln können. Danach können Sie zum Beispiel nicht mehr `join()` aufrufen, weil die Variable keinen Thread mehr repräsentiert.
Sie können in einem Thread grundsätzlich all das machen, was Sie in jeder beliebigen Funktion machen können. Letztendlich ist der Thread nichts anderes als eine Funktion – mit der Besonderheit, dass sie zeitgleich zu anderen Funktionen ausgeführt wird. Im Beispiel 44.1 werden in einer Schleife fünf Zahlen auf die Standardausgabe ausgegeben. Damit die Datenausgabe nicht zu schnell erfolgt, wird in jedem Schleifendurchgang die Funktion `wait()` aufgerufen, mit der die Ausführung des aktuellen Threads für jeweils eine Sekunde angehalten wird.
Die Funktion `wait()` greift dazu auf eine freistehende Funktion namens `sleep_for()` zu, die ebenfalls von Boost.Thread stammt und im Namensraum `boost::this_thread` definiert ist.
Sie müssen `sleep_for()` angeben, für wie lange der aktuelle Thread angehalten werden soll. Indem ein Objekt vom Typ `boost::chrono::seconds` übergeben wird, wird eine Zeitspanne definiert, für die der aktuelle Thread angehalten wird. `boost::chrono::seconds` stammt aus der Bibliothek Boost.Chrono, die im Kapitel 37 vorgestellt wird.
Beachten Sie, dass `sleep_for()` ausschließlich Zeitangaben aus der Bibliothek Boost.Chrono akzeptiert. Auch wenn Boost.Chrono in die C++11-Standardbibliothek eingegangen ist – Zeitangaben aus dem Namensraum `std::chrono` werden nicht akzeptiert und führen zu Compiler-Fehlern.
Möchten Sie `join()` nicht explizit aufrufen, um am Ende der Funktion `main()` auf den Thread zu warten, können Sie die Klasse `boost::scoped_thread` verwenden.

Beispiel 44.2 Mit `boost::scoped_thread` auf einen Thread warten

```
#include <boost/thread.hpp>
#include <boost/thread/scoped_thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << '\n';
  }
}

int main()
{
  boost::scoped_thread<> t{boost::thread{thread}};
}
```

Sie übergeben an den Konstruktor von `boost::scoped_thread` einen Thread in Form einer Instanz von `boost::thread`. Der Thread wird wie im Beispiel zuvor gestartet. Die Besonderheit von `boost::scoped_thread` liegt darin, dass diese Klasse im Destruktor eine Aktion ausführt, in der auf den Thread zugegriffen werden kann. Standardmäßig ruft `boost::scoped_thread` für den Thread `join()` auf. Somit verhält sich Beispiel 44.2 wie das vorherige.
Sie können eigene Aktionen als Template-Parameter übergeben. Dabei muss es sich um eine Klasse handeln, die den Operator `operator()` derart überlädt, dass er einen Parameter vom Typ `boost::thread` erwartet. `boost::scoped_thread` garantiert, dass der Operator im Destruktor aufgerufen wird.
`boost::scoped_thread` ist eine Klasse, die Sie ausschließlich in Boost.Thread finden. Es gibt in der Standard-

bibliothek kein Gegenstück. Beachten Sie, dass Sie für `boost::scoped_thread` jedoch zusätzlich die Headerdatei `boost/thread/scoped_thread.hpp` einbinden müssen.
Im Folgenden lernen Sie *Unterbrechungspunkte* kennen, mit Hilfe derer ein Thread einen anderen unterbrechen kann. Unterbrechungspunkte werden ebenfalls aussschließlich von Boost.Thread unterstützt und fehlen in der Standardbibliothek.

Beispiel 44.3 Ein Unterbrechungspunkt mit `boost::this_thread::sleep_for()`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << '\n';
    }
  }
  catch (boost::thread_interrupted&) {}
}

int main()
{
  boost::thread t{thread};
  wait(3);
  t.interrupt();
  t.join();
}
```

Wenn Sie die Methode `interrupt()` aufrufen, wird der entsprechende Thread unterbrochen. Unterbrochen bedeutet, dass im Thread eine Ausnahme vom Typ `boost::thread_interrupted` geworfen wird. Dies geschieht jedoch nur dann, wenn der Thread einen Unterbrechungspunkt erreicht.
Der alleinige Aufruf von `interrupt()` bewirkt nichts, wenn ein Thread keine Unterbrechungspunkte enthält. Beim Aufruf von `interrupt()` wird also nicht sofort eine Ausnahme vom Typ `boost::thread_interrupted` geworfen. Stattdessen überprüft der Thread in den bereits erwähnten Unterbrechungspunkten, ob `interrupt()` aufgerufen wurde – ist dies der Fall, wird eine Ausnahme vom Typ `boost::thread_interrupted` geworfen.
Boost.Thread definiert eine Reihe von Unterbrechungspunkten. Ein Unterbrechungspunkt ist zum Beispiel `sleep_for()`. Da `sleep_for()` aufgrund der Schleife in `thread()` fünfmal aufgerufen wird, überprüft der Thread fünfmal, ob er unterbrochen wurde. Zwischen den Aufrufen von `sleep_for()` kann der Thread im Beispiel 44.3 nicht unterbrochen werden, und es wird keine Ausnahme vom Typ `boost::thread_interrupted` geworfen.
Wenn Sie Beispiel 44.3 ausführen, stellen Sie fest, dass nicht mehr alle fünf Zahlen ausgegeben werden. Der Grund ist, dass in der Funktion `main()` nach drei Sekunden `interrupt()` aufgerufen wird. Damit wird `thread()` nach drei Sekunden unterbrochen, und es wird eine Ausnahme vom Typ `boost::thread_interrupted` geworfen. Diese wird zwar im Thread abgefangen, ohne dass etwas im `catch`-Block geschieht. Weil jedoch anschließend die Funktion `thread()` zurückkehrt, endet der Thread – und damit das Programm, da `main()` lediglich mit `join()` auf das Ende des Threads wartet.
Boost.Thread definiert rund fünfzehn Unterbrechungspunkte, zu denen wie eben gesehen die Funktion `sleep_for()` zählt. Dank dieser Unterbrechungspunkte lassen sich relativ einfach und zeitnah Threads unterbrechen. Dadurch, dass immer erst ein Unterbrechungspunkt erreicht werden muss, um zu überprüfen, ob eine Ausnahme vom Typ `boost::thread_interrupted` geworfen werden muss, sind Unterbrechungspunkte aber nicht zwangsläufig die jeweils beste Wahl.

Beispiel 44.4 Unterbrechungspunkte mit `disable_interruption` deaktivieren

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  boost::this_thread::disable_interruption no_interruption;
  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << '\n';
    }
  }
  catch (boost::thread_interrupted&) {}
}

int main()
{
  boost::thread t{thread};
  wait(3);
  t.interrupt();
  t.join();
}
```

Mit `boost::this_thread::disable_interruption` bietet Boost.Thread eine Klasse an, mit der verhindert werden kann, dass ein Thread unterbrochen wird. Wenn Sie eine Instanz dieser Klasse erstellen, sind Unterbrechungspunkte im aktuellen Thread so lange deaktiviert, so lange die Instanz existiert. Beispiel 44.4 gibt dementsprechend fünf Zahlen aus, da der Versuch, den Thread zu unterbrechen, ignoriert wird.

Beispiel 44.5 Thread-Eigenschaften über `boost::thread::attributes` setzen

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{

  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << '\n';
    }
  }
  catch (boost::thread_interrupted&) {}
}

int main()
{
```

```
  boost::thread::attributes attrs;
  attrs.set_stack_size(1024);
  boost::thread t{attrs, thread};
  t.join();
}
```

Über `boost::thread::attributes` ist es möglich, Thread-Eigenschaften zu setzen. Genaugenommen kann jedoch nur eine einzige Thread-Eigenschaft portabel gesetzt werden, nämlich die Größe des Stacks. Im Beispiel 44.5 wird dieser über `boost::thread::attributes::set_stack_size()` auf 1024 Bytes gesetzt.

Beispiel 44.6 Thread-ID und Anzahl verfügbarer Prozessoren/Kerne erhalten

```
#include <boost/thread.hpp>
#include <iostream>

int main()
{
  std::cout << boost::this_thread::get_id() << '\n';
  std::cout << boost::thread::hardware_concurrency() << '\n';
}
```

Über den Namensraum `boost::this_thread` können wie im Beispiel 44.6 verschiedene freistehende Funktionen aufgerufen werden, die sich auf den aktuellen Thread beziehen. `sleep_for()` hatten Sie bereits kennengelernt. Eine weitere Funktionen ist `get_id()`: Sie gibt eine Nummer zurück, mit der der aktuelle Thread identifiziert werden kann. `get_id()` steht auch als Methode der Klasse `boost::thread` zur Verfügung.
Die Methode `hardware_concurrency()`, die als statische Methode zur Klasse `boost::thread` gehört, gibt die Zahl der Threads zurück, die tatsächlich dank verfügbarer Prozessoren oder Prozessorkerne gleichzeitig ausgeführt werden können. So wird zum Beispiel auf einem Computer mit Dual-Core-Prozessor der Wert 2 zurückgegeben. Der Rückgabewert erlaubt es einem Programm zum Beispiel, genau so viele Threads zu erstellen, wie von der Hardware, auf der ein Programm läuft, gleichzeitig ausgeführt werden können.
Boost.Thread bietet mit `boost::thread_group` außerdem eine Klasse an, um Threads in einer Gruppe zusammenzufassen und gemeinsam zu verwalten. So stellt diese Klasse zum Beispiel die Methode `join_all()` zur Verfügung, um auf alle Threads in der Gruppe zu warten.

44.2 Threads synchronisieren

Während der Einsatz mehrerer Threads die Performance eines Programms erhöhen kann, erhöht sich üblicherweise auch die Komplexität. Wenn mehrere Funktionen dank Threads gleichzeitig ausgeführt werden, müssen Zugriffe dieser Threads auf globale Ressourcen, die also mehreren Threads zur Verfügung stehen, synchronisiert werden. Die Synchronisation von Threads kann in größeren Programmen sehr schwierig sein und viel Detailarbeit erfordern. Im Folgenden lernen Sie die Klassen kennen, die Boost.Thread zur Synchronisation von Threads zur Verfügung stellt.

Beispiel 44.7 Exklusiver Zugriff mit `boost::mutex`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::mutex mutex;

void thread()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
```

```
    mutex.lock();
    std::cout << "Thread " << get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1{thread};
  boost::thread t2{thread};
  t1.join();
  t2.join();
}
```

Multithreaded-Programme verwenden zur Synchronisation Objekte, die *Mutex* genannt werden. Boost.Thread stellt entsprechend verschiedene Mutex-Klassen zur Verfügung, von denen die einfachste `boost::mutex` ist. Das Grundprinzip eines Mutex ist: Wenn ein Thread einen Mutex in Besitz nimmt, können andere Threads den gleichen Mutex erst dann ihrerseits in Besitz nehmen, wenn der Mutex wieder freigegeben wurde. Auf diese Weise können Threads gezwungen werden zu warten, bis ein anderer Thread bestimmte Operationen ausgeführt und anschließend einen Mutex wieder freigegeben hat.
Im Beispiel 44.7 wird ein globales Objekt **`mutex`** vom Typ `boost::mutex` verwendet. Dieser Mutex wird innerhalb der `for`-Schleife in der Funktion `thread()` kurz vor dem Zugriff auf die Standardausgabe in Besitz genommen. Dies geschieht über den Aufruf der Methode `lock()`. Nachdem eine Meldung auf die Standardausgabe ausgegeben wurde, wird der Mutex mit `unlock()` wieder freigegeben.
In der Funktion `main()` wird die Funktion `thread()` in zwei Threads gestartet. Jeder dieser beiden Threads zählt innerhalb einer `for`-Schleife bis fünf und gibt in jedem Schleifendurchgang eine Meldung auf die Standardausgabe aus. Da die Standardausgabe ein globales Objekt ist, das von beiden Threads geteilt wird, muss der Zugriff auf **`std::cout`** synchronisiert werden. Andernfalls könnten sich Schreibvorgänge überschneiden. Die Synchronisierung stellt sicher, dass zu jedem beliebigen Zeitpunkt nur ein einziger Thread auf **`std::cout`** zugreift.
Indem beide Threads vor dem Zugriff auf die Standardausgabe versuchen, den gleichen Mutex in Besitz zu nehmen, ist garantiert, dass genau ein Thread auf die Standardausgabe zugreift. Egal, welcher Thread erfolgreich `lock()` aufruft – der andere Thread muss warten, bis `unlock()` aufgerufen wurde.
Diese für Mutexe typische Vorgehensweise – das In-Besitz-nehmen und wieder Freigeben – wird durch verschiedene Typen in Boost.Thread unterstützt. Anstatt `lock()` und `unlock()` selbst aufzurufen, kann zum Beispiel die Klasse `boost::lock_guard` verwendet werden.

Beispiel 44.8 `boost::lock_guard` mit garantierter Mutex-Freigabe

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::mutex mutex;

void thread()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::lock_guard<boost::mutex> lock{mutex};
    std::cout << "Thread " << get_id() << ": " << i << std::endl;
  }
}

int main()
{
```

```
  boost::thread t1{thread};
  boost::thread t2{thread};
  t1.join();
  t2.join();
}
```

Im Beispiel 44.8 wird für den Mutex automatisch `lock()` im Konstruktor und `unlock()` im Destruktor von `boost::lock_guard` aufgerufen. Der Zugriff ist wie zuvor beim expliziten Aufruf von `lock()` und `unlock()` synchronisiert. Die Klasse `boost::lock_guard` ist ein Beispiel für das RAII-Idiom, mit dessen Hilfe sichergestellt werden kann, dass Ressourcen garantiert freigegeben werden.
Neben `boost::mutex` und `boost::lock_guard` bietet Boost.Thread weitere Klassen an, die verschiedene Spielarten der Synchronisierung unterstützen. Eine wichtige Klasse ist dabei `boost::unique_lock`, die im Vergleich zu `boost::lock_guard` eine Reihe nützlicher Methoden bietet und nicht nur aus einem Konstruktor und Destruktor besteht.

Beispiel 44.9 Der Allrounder-Lock `boost::unique_lock`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::timed_mutex mutex;

void thread1()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::unique_lock<boost::timed_mutex> lock{mutex};
    std::cout << "Thread " << get_id() << ": " << i << std::endl;
    boost::timed_mutex *m = lock.release();
    m->unlock();
  }
}

void thread2()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::unique_lock<boost::timed_mutex> lock{mutex,
      boost::try_to_lock};
    if (lock.owns_lock() || lock.try_lock_for(boost::chrono::seconds{1}))
    {
      std::cout << "Thread " << get_id() << ": " << i << std::endl;
    }
  }
}

int main()
{
  boost::thread t1{thread1};
  boost::thread t2{thread2};
  t1.join();
  t2.join();
}
```

Im Beispiel 44.9 kommen zwei Varianten der Funktion `thread()` vor, wie sie in den vorherigen Beispielen eingesetzt wurde. Beide Varianten geben noch immer in einer Schleife fünf Zahlen auf die Standardausgabe aus. Beide Varianten verwenden jedoch nun die Klasse `boost::unique_lock`, um einen Mutex in Besitz zu nehmen.
In `thread1()` wird **`mutex`** an den Konstruktor von `boost::unique_lock` übergeben. `boost::unique_lock` versucht daraufhin, den Mutex in Besitz zu nehmen. In diesem Fall verhält sich `boost::unique_lock` nicht anders als `boost::lock_guard`. `boost::unique_lock` ruft im Konstruktor `lock()` für den Mutex auf.
Der Mutex wird jedoch nicht im Destruktor von `boost::unique_lock` freigegeben. In `thread1()` wird für den Lock `release()` aufgerufen. Dadurch wird der Mutex vom Lock entkoppelt. So wie `boost::lock_guard` gibt `boost::unique_lock` üblicherweise im Destruktor einen Mutex frei. Wird der Mutex entkoppelt, geschieht dies jedoch nicht. Der Mutex wird daher in `thread1()` explizit über einen Aufruf von `unlock()` freigegeben.
In `thread2()` wird an den Konstruktor von `boost::unique_lock` neben **`mutex`** **`boost::try_to_lock`** übergeben. In diesem Fall ruft der Konstruktor von `boost::unique_lock` für den Mutex nicht `lock()` auf, sondern `try_lock()`. Der Lock versucht also lediglich, den Mutex in Besitz zu nehmen. Ist der Mutex von einem anderen Thread in Besitz genommen worden, schlägt der Versuch fehl.
Über `owns_lock()` kann überprüft werden, ob `boost::unique_lock` einen Mutex in Besitz genommen hat. Gibt `owns_lock()` `true` zurück, kann im `thread2()` sofort auf **`std::cout`** zugegriffen werden.
Gibt `owns_lock()` `false` zurück, wird die Methode `try_lock_for()` aufgerufen. Diese Methode versucht ebenfalls, den Mutex in Besitz zu nehmen. Der Versuch wird jedoch erst nach Ablauf einer bestimmten Zeitspanne abgebrochen. Im Beispiel 44.9 ist angegeben, dass **`lock`** eine Sekunde lang versuchen soll, den Mutex in Besitz zu nehmen. Gibt `try_lock_for()` `true` zurück, kann auf **`std::cout`** zugegriffen werden. Andernfalls gibt `thread2()` auf und überspringt die Ausgabe. Wenn Sie das Beispiel ausführen, ist es möglich, dass der zweite Thread weniger als fünf Zahlen ausgibt.
Beachten Sie, dass der Typ von **`mutex`** im Beispiel 44.9 nicht `boost::mutex`, sondern `boost::timed_mutex` ist. Im Beispiel muss diese Klasse verwendet werden, da nur sie die Methode `try_lock_for()` anbietet. Auf diese Methode greift der Lock zu, wenn für diesen `try_lock_for()` aufgerufen wird. `boost::mutex` bietet lediglich die Methoden `lock()` und `try_lock()` an.
Die Klasse `boost::unique_lock` ist ein *exklusiver Lock*. Das bedeutet, dass jeweils nur ein Thread mit dieser Klasse einen Mutex in Besitz nehmen kann und andere Threads warten müssen, bis der Mutex wieder freigegeben wurde. Neben exklusiven Locks gibt es auch *nicht-exklusive Locks*. Boost.Thread bietet hierfür die Klasse `boost::shared_lock` an, die zusammen mit einem Mutex vom Typ `shared_mutex` verwendet werden muss.

Beispiel 44.10 Nicht-exklusive Locks mit `boost::shared_lock`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::shared_mutex mutex;
std::vector<int> random_numbers;

void fill()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::shared_mutex> lock{mutex};
    random_numbers.push_back(std::rand());
    lock.unlock();
    wait(1);
  }
}

void print()
```

```
{
  for (int i = 0; i < 3; ++i)
  {
    wait(1);
    boost::shared_lock<boost::shared_mutex> lock{mutex};
    std::cout << random_numbers.back() << '\n';
  }
}

int sum = 0;

void count()
{
  for (int i = 0; i < 3; ++i)
  {
    wait(1);
    boost::shared_lock<boost::shared_mutex> lock{mutex};
    sum += random_numbers.back();
  }
}

int main()
{
  boost::thread t1{fill}, t2{print}, t3{count};
  t1.join();
  t2.join();
  t3.join();
  std::cout << "Sum: " << sum << '\n';
}
```

Nicht-exklusive Locks vom Typ `boost::shared_lock` können dann verwendet werden, wenn Threads lediglich lesend auf eine Ressource zugreifen. Ein Thread, der eine Ressource verändert und daher schreibend auf sie zugreift, benötigt einen exklusiven Lock. Das sollte einleuchten: Threads, die lediglich lesend auf eine Ressource zugreifen, merken nicht, dass eine Ressource zeitgleich in einem anderen Thread gelesen wird. Nicht-exklusive Locks können daher einen Mutex mit anderen nicht-exklusiven Locks teilen.
Im Beispiel 44.10 greifen die beiden Funktionen `print()` und `count()` lesend auf **`random_numbers`** zu. Während `print()` die letzte Zahl in **`random_numbers`** auf die Standardausgabe ausgibt, addiert `count()` sie zur Variablen **`sum`** hinzu. Weil beide Funktionen **`random_numbers`** nicht ändern, können sie gleichzeitig auf diese Variable zugreifen. Deswegen wird der Zugriff auf **`random_numbers`** mit einem nicht-exklusiven Lock vom Typ `boost::shared_lock` synchronisiert.
In der Funktion `fill()` jedoch wird ein exklusiver Lock vom Typ `boost::unique_lock` benötigt, da in dieser Funktion neue Zufallszahlen in den Container **`random_numbers`** eingefügt werden. Damit der Mutex freigegeben wird, bevor in der `for`-Schleife der Funktion `fill()` eine Sekunde gewartet wird, wird explizit `unlock()` aufgerufen. Die Funktion `wait()` wird im Gegensatz zu den vorherigen Beispielen nicht zu Beginn, sondern am Ende der `for`-Schleife aufgerufen, damit der Container **`random_numbers`** auf alle Fälle eine Zufallszahl enthält, bevor `print()` und `count()` zum ersten Mal auf den Container zugreifen. In diesen Funktionen wird `wait()` zu Beginn der `for`-Schleifen aufgerufen.
Wenn Ihnen die Aufrufe von `wait()` an den unterschiedlichen Stellen in den `for`-Schleifen nicht geheuer sind, haben Sie Recht: Je nachdem, welche Threads wann und wie schnell vom Prozessor ausgeführt werden, kann die Reihenfolge durcheinander geraten. Mit Hilfe von *Bedingungsvariablen* können die Threads so synchronisiert werden, dass Zahlen sofort dann, wenn sie dem Container **`random_numbers`** hinzugefügt wurden, in einem anderen Thread verarbeitet werden.

Beispiel 44.11 Bedingungsvariablen mit `boost::condition_variable_any`

```
#include <boost/thread.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

boost::mutex mutex;
boost::condition_variable_any cond;
```

```
std::vector<int> random_numbers;

void fill()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::mutex> lock{mutex};
    random_numbers.push_back(std::rand());
    cond.notify_all();
    cond.wait(mutex);
  }
}

void print()
{
  std::size_t next_size = 1;
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::mutex> lock{mutex};
    while (random_numbers.size() != next_size)
      cond.wait(mutex);
    std::cout << random_numbers.back() << '\n';
    ++next_size;
    cond.notify_all();
  }
}

int main()
{
  boost::thread t1{fill};
  boost::thread t2{print};
  t1.join();
  t2.join();
}
```

Im Beispiel 44.11 wurden die Funktionen `wait()` und `count()` aus dem vorherigen Beispiel entfernt. Threads warten also nicht mehr pro Schleifendurchgang eine Sekunde, sondern arbeiten so schnell wie möglich. Außerdem wird keine Summe mehr gebildet – Zahlen werden lediglich auf die Standardausgabe ausgegeben.
Damit die Verarbeitung der Zufallszahlen nicht durcheinander kommt, müssen die Threads mit Hilfe von Bedingungsvariablen synchronisiert werden. Diese ermöglichen es, Bedingungen thread-übergreifend zu überprüfen.
Die Funktion `fill()` generiert wie zuvor pro Schleifendurchgang eine Zufallszahl und speichert sie im Container **`random_numbers`**. Dazu muss wie zuvor ein exklusiver Lock verwendet werden, damit ein anderer Thread nicht gleichzeitig auf **`random_numbers`** zugreift, während in diesen Container eine neue Zufallszahl eingefügt wird. Am Ende der `for`-Schleife in der Funktion `fill()` wird nicht mehr einfach nur eine Sekunde gewartet, sondern auf eine Bedingungsvariable zugegriffen und die Methode `notify_all()` aufgerufen. Diese Methode weckt alle Threads auf, in denen für die gleiche Bedingungsvariable `wait()` aufgerufen wurde und die daher auf eine Benachrichtigung, wie sie durch `notify_all()` ausgelöst wird, warten.
Wenn Sie sich die `for`-Schleife in der Funktion `print()` ansehen, stellen Sie fest, dass dort für die gleiche Bedingungsvariable, die in der Funktion `fill()` verwendet wird, `wait()` aufgerufen wird. Wenn der Thread durch einen Aufruf von `notify_all()` geweckt wird, versucht er, den Mutex in Besitz zu nehmen. Dies gelingt erst, nachdem der Mutex in der Funktion `fill()` freigegeben wurde.
Der Trick ist, dass der Aufruf von `wait()` gleichzeitig den entsprechenden Mutex freigibt, der als Parameter übergeben wird. Da am Ende der `for`-Schleife in der Funktion `fill()` für die gleiche Bedingungsvariable, für die gerade `notify_all()` aufgerufen wurde, `wait()` aufgerufen wird, wird der Mutex freigegeben. Das heißt, die Funktion `fill()` wartet, bis jemand anderes für die Bedingungsvariable `notify_all()` aufruft. Dies geschieht in `print()`, nachdem in dieser Funktion die neue Zufallszahl auf die Standardausgabe ausgegeben wurde.
Beachten Sie, dass der Aufruf von `wait()` in der Funktion `print()` außerdem in einer `while`-Schleife erfolgt: Es könnte sein, dass der Thread, der die Funktion `fill()` ausführt, zuerst eine Zufallszahl im Container **`random_numbers`** speichert, bevor in `print()` `wait()` für die Bedingungsvariable aufgerufen wird. Damit die

Funktion `print()` trotzdem die erste bereits zu **`random_numbers`** hinzugefügte Zufallszahl verarbeitet, wird die Anzahl der im Container gespeicherten Zufallszahlen mit dem nächsten erwarteten Wert verglichen.
Beispiel 44.11 funktioniert auch problemlos, wenn Sie die Locks nicht in, sondern vor die `for`-Schleifen setzen. Das ergibt genaugenommen mehr Sinn, da in diesem Fall die Locks nicht in jedem Schleifendurchgang initialisiert und zerstört werden. Da der Mutex jeweils beim Aufruf von `wait()` freigegeben wird, ist es nicht notwendig, die Locks am Ende jedes Schleifendurchgangs zu zerstören, um auf diese Weise den Mutex freizugeben.

44.3 Thread-spezifischer Speicher

Thread-spezifischer Speicher – auf Englisch thread local storage (TLS) – ist ein Speicherbereich, auf den jeweils nur ein Thread Zugriff hat. Sie können sich TLS-Variablen als globale Variablen vorstellen, die jedoch nicht global im gesamten Prozess sind, sondern lediglich in einem Thread. Welchen Nutzen derartige Variablen haben, soll Ihnen anhand des folgenden Beispiels gezeigt werden.

Beispiel 44.12 Synchronisation mehrerer Threads mit einer statischen Variablen

```
#include <boost/thread.hpp>
#include <iostream>

boost::mutex mutex;

void init()
{
  static bool done = false;
  boost::lock_guard<boost::mutex> lock{mutex};
  if (!done)
  {
    done = true;
    std::cout << "done" << '\n';
  }
}

void thread()
{
  init();
  init();
}

int main()
{
  boost::thread t[3];

  for (int i = 0; i < 3; ++i)
    t[i] = boost::thread{thread};

  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

Beispiel 44.12 führt die Funktion `thread()` in drei Threads aus. `thread()` ruft eine andere Funktionen `init()` zweimal auf. `init()` überprüft, ob die Variable **done** vom Typ bool `false` ist. Ist sie das, wird sie auf `true` gesetzt und `done` auf die Standardausgabe ausgegeben.
Da **done** eine statische Variable ist, wird sie von allen Threads geteilt. Wenn **done** im ersten Thread auf `true` gesetzt wird, wird der zweite und dritte Thread kein `done` auf die Standardausgabe ausgeben. Der zweite Aufruf von `init()` in `thread()` führt ebenfalls nicht zu einer Ausgabe von `done`. Beispiel 44.12 gibt `done` genau einmal auf die Standardausgabe aus.
Eine statische Variable wie **done** kann verwendet werden, um eine einmalige Initialisierung vorzunehmen – einmal in einem Prozess und unabhängig davon, wie viele Threads verwendet werden. Soll eine einmalige Initialisierung pro Thread stattfinden, können TLS-Variablen verwendet werden.

Beispiel 44.13 Synchronisation mehrerer Threads mit TLS-Variablen

```
#include <boost/thread.hpp>
```

```
#include <iostream>

boost::mutex mutex;

void init()
{
  static boost::thread_specific_ptr<bool> tls;
  if (!tls.get())
  {
    tls.reset(new bool{true});
    boost::lock_guard<boost::mutex> lock{mutex};
    std::cout << "done" << '\n';
  }
}

void thread()
{
  init();
  init();
}

int main()
{
  boost::thread t[3];

  for (int i = 0; i < 3; ++i)
    t[i] = boost::thread{thread};

  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

Im Beispiel 44.13 wurde die statische Variable **done** durch eine TLS-Variable **tls** ersetzt. Sie basiert auf der Template-Klasse `boost::thread_specific_ptr`, die mit dem Typ bool instanziiert ist. Die neue Variable **tls** funktioniert grundsätzlich genauso wie **done**: Es handelt sich um einen Schalter, mit dem kontrolliert wird, ob etwas bereits erledigt wurde. Der entscheidende Unterschied ist, dass der Wert, den **tls** speichert, nur im jeweiligen Thread verfügbar ist. Auch wenn die statische Variable **tls** nur einmal existiert, so existiert ein in **tls** gespeicherter Wert nur im jeweiligen Thread und ist für andere Threads nicht sichtbar.
Nachdem eine Variable vom Typ `boost::thread_specific_ptr` erstellt wurde, kann sie gesetzt werden. Die Klasse `boost::thread_specific_ptr` erwartet jedoch keine bool-Variable, sondern die Adresse einer bool-Variablen. Indem die Methode `reset()` aufgerufen wird, kann die Adresse einer bool-Variablen in **tls** gespeichert werden. Im Beispiel 44.13 wird eine bool-Variable dynamisch reserviert und die Adresse, die von `new` zurückgegeben wird, in **tls** gespeichert. Damit dies nur einmal geschieht, wird vorher mit `get()` überprüft, ob bereits eine Adresse in der Variablen **tls** gespeichert ist.
Da `boost::thread_specific_ptr` eine Adresse speichert, verhält sich diese Klasse wie ein Zeiger. So stehen zum Beispiel die Operatoren `operator*` und `operator->` zur Verfügung, die genauso funktionieren wie von Zeigern gewohnt.
Beispiel 44.13 gibt `done` dreimal auf die Standardausgabe aus. Jeder Thread gibt `done` einmal aus, und zwar jeweils im ersten Aufruf von `init()`. Weil das Beispiel eine TLS-Variable verwendet, besitzt jeder Thread seinen eigenen Schalter. Wenn der erste Thread die Variable **tls** mit einem Zeiger auf eine dynamisch reservierte bool-Variable initialisiert, ist **tls** in den anderen beiden Threads noch nicht initialisiert. Weil TLS-Variablen nicht global im gesamten Prozess, sondern nur global in einem Thread sind, hat eine Änderung der Variablen **tls** in einem Thread keine Auswirkungen auf andere Threads.

44.4 Futures und Promises

Futures und Promises stellen einen Mechanismus dar, Daten von einem Thread an einen anderen zu übergeben. Während dies auch zum Beispiel über globale Variablen bewerkstelligt werden kann, kommt die Datenübergabe mit Futures und Promises ohne globale Variablen aus. Darüberhinaus müssen Sie sich nicht selbst um eine Synchronisation kümmern.

Ein *Future* ist eine Variable, die einen Wert erhalten wird, der von einem anderen Thread berechnet wird. Wenn Sie auf ein Future zugreifen, um die Variable auszulesen, müssen Sie womöglich warten, bis der entsprechende Thread den Wert errechnet hat. Boost.Thread stellt den Typ `boost::future` zur Verfügung, um ein Future zu definieren. Die Klasse bietet eine Methode `get()` an, um den Wert der Variablen zu erhalten. Diese Methode ist blockierend, da sie unter Umständen auf einen Thread warten muss.
Um ein Future auf einen Wert zu setzen, muss auf ein mit dem Future verbundenes *Promise* zugegriffen werden. Während `boost::future` eine Methode `get()` besitzt, existiert keine Methode, um einen Wert zu setzen.
Für Promises stellt Boost.Thread die Klasse `boost::promise` bereit. Diese Klasse besitzt eine Methode `set_value()`. Der Trick ist, dass Futures und Promises als Paar auftreten. So kann über die Methode `get_future()` ein Future von einem Promise erhalten werden. Future und Promise können in unterschiedlichen Threads verwendet werden. Wird das Promise in einem Thread gesetzt, kann der entsprechende Wert über den Future im anderen Thread gelesen werden.

Beispiel 44.14 `boost::future` und `boost::promise` in Aktion

```
#define BOOST_THREAD_PROVIDES_FUTURE
#include <boost/thread.hpp>
#include <boost/thread/future.hpp>
#include <functional>
#include <iostream>

void accumulate(boost::promise<int> &p)
{
  int sum = 0;
  for (int i = 0; i < 5; ++i)
    sum += i;
  p.set_value(sum);
}

int main()
{
  boost::promise<int> p;
  boost::future<int> f = p.get_future();
  boost::thread t{accumulate, std::ref(p)};
  std::cout << f.get() << '\n';
}
```

Im Beispiel 44.14 kommen ein Future und ein Promise zum Einsatz. Das Future **f** wird über `get_future()` vom Promise **p** erhalten. Daraufhin wird eine Referenz auf den Promise an den Thread **t** übergeben, der die Funktion `accumulate()` ausführt. In `accumulate()` wird die Summe der Zahlen 1 bis 5 errechnet und dann im Promise gespeichert. In `main()` wird lediglich `get()` für den Future aufgerufen, um den entsprechenden Wert in die Standardausgabe zu schreiben.
Das Future **f** und das Promise **p** sind verknüpft. Wenn für das Future `get()` aufgerufen wird, wird der Wert zurückgegeben, der mit `set_value()` im Promise gesetzt wurde. Da das Beispiel aus zwei Threads besteht, ist es möglich, dass `get()` in `main()` aufgerufen wird, bevor in `acccumulate()` `set_value()` aufgerufen wurde. In diesem Fall blockiert `get()` und kehrt erst zurück, wenn mit `set_value()` ein Wert im Promise gespeichert wurde.
Wenn Sie Beispiel 44.14 ausführen, wird `10` ausgegeben.
Das Beispiel erfordert, dass die Funktion `accumulate()` für den Einsatz in einem Thread angepasst wurde. So muss die Funktion einen Parameter vom Typ `boost::promise` erwarten und in diesem das Ergebnis speichern. Im folgenden Beispiel lernen Sie die Klasse `boost::packaged_task` kennen, mit der Sie jede beliebige Funktion verwenden können, die das Ergebnis mit `return` zurückgibt.

Beispiel 44.15 `boost::packaged_task` in Aktion

```
#define BOOST_THREAD_PROVIDES_FUTURE
#include <boost/thread.hpp>
#include <boost/thread/future.hpp>
#include <utility>
#include <iostream>

int accumulate()
{
  int sum = 0;
```

```
  for (int i = 0; i < 5; ++i)
    sum += i;
  return sum;
}

int main()
{
  boost::packaged_task<int> task{accumulate};
  boost::future<int> f = task.get_future();
  boost::thread t{std::move(task)};
  std::cout << f.get() << '\n';
}
```

Beispiel 44.15 ist vergleichbar mit dem vorherigen. Diesmal wird jedoch kein Promise vom Typ `boost::promise` verwendet. Stattdessen kommt `boost::packaged_task` zum Einsatz. Diese Klasse bietet wie `boost::promise` eine Methode `get_future()` an, um ein Future zu erhalten.
Dem Konstruktor von `boost::packaged_task` muss eine Funktion übergeben werden, die in einem Thread ausgeführt werden soll. `boost::packaged_task` selbst startet keinen Thread. Ein Objekt vom Typ `boost::packaged_task` muss an den Konstruktor von `boost::thread` übergeben werden, damit der Thread startet und die entsprechende Funktion im Thread ausgeführt wird.
Der Vorteil von `boost::packaged_task` ist, dass der Rückgabewert der entsprechenden Funktion im Future gesetzt wird. Es ist nicht notwendig, eine Funktion dahingehend zu ändern, dass sie ein Ergebnis in einem Promise speichert. `boost::packaged_task` ist in gewisser Weise ein Adapter, der den Rückgabewert einer Funktion in einem Future ablegt.
Während im obigen Beispiel `boost::promise` weggelassen werden konnte, wird im folgenden zusätzlich auf `boost::packaged_task` und `boost::thread` verzichtet.

Beispiel 44.16 `boost::async()` in Aktion

```
#define BOOST_THREAD_PROVIDES_FUTURE
#include <boost/thread.hpp>
#include <boost/thread/future.hpp>
#include <iostream>

int accumulate()
{
  int sum = 0;
  for (int i = 0; i < 5; ++i)
    sum += i;
  return sum;
}

int main()
{
  boost::future<int> f = boost::async(accumulate);
  std::cout << f.get() << '\n';
}
```

Im Beispiel 44.16 wird `accumulate()` an die Funktion `boost::async()` übergeben. Diese Funktion vereint `boost::packaged_task` und `boost::thread`. Sie startet `accumulate()` in einem Thread und gibt ein Future vom Typ `boost::future` zurück.
Sie können `boost::async()` eine *Launch-Policy* übergeben. Dieser zusätzliche Parameter bestimmt, ob `boost::async()` die Funktion in einem neuen oder im aktuellen Thread ausführt. Wenn Sie **`boost::launch::async`** als ersten Parameter an `boost::async()` übergeben, wird ein neuer Thread gestartet. Übergeben Sie **`boost::launch::deferred`**, wird die Funktion im aktuellen Thread ausgeführt. **`boost::launch::async`** ist der Standardwert, der gilt, wenn Sie keine Launch-Policy angeben.
In Boost 1.57.0 fehlt die Implementation für **`boost::launch::deferred`**. Übergeben Sie **`boost::launch::deferred`** an **`boost::launch::async`**, wird Ihr Programm sofort beendet.

Kapitel 45

Boost.Atomic

Boost.Atomic bietet eine Klasse `boost::atomic` an, mit der atomare Variablen erstellt werden können. Atomare Variablen heißen so, weil Zugriffe auf diese atomar sind. Boost.Atomic wird in Multithreaded-Programmen verwendet, wenn der Zugriff auf eine Variable in einem Thread nicht von einem Zugriff auf dieselbe Variable in einem anderen Thread unterbrochen werden können soll. Ohne `boost::atomic` müssten Zugriffe auf von Threads geteilte Variablen mit Locks synchronisiert werden.
Beachten Sie, dass `boost::atomic` darauf angewiesen ist, dass Zielplattformen atomare Variablenzugriffe unterstützen. Andernfalls greift `boost::atomic` auf Locks zu. Die Bibliothek bietet eine Möglichkeit herauszufinden, ob atomare Variablenzugriffe auf einer Zielplattform unterstützt werden.
Wenn Ihre Entwicklungsumgebung C++11 unterstützt, können Sie Boost.Atomic links liegen lassen. Die C++11-Standardbibliothek enthält eine Headerdatei `atomic`, die die gleichen Klassen und Funktionen wie Boost.Atomic zur Verfügung stellt. So finden Sie dort zum Beispiel eine Klasse `std::atomic`.
Boost.Atomic unterstützt annähernd den gleichen Funktionsumfang wie die Standardbibliothek. Während einige Funktionen in Boost.Atomic mehrfach überladen sind, können sie in der Standardbibliothek unterschiedliche Namen haben. Mit `std::kill_dependency()` unterstützt die Standardbibliothek außerdem eine Funktion, die in Boost.Atomic fehlt.

Beispiel 45.1 `boost::atomic` in Aktion

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  ++a;
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Im Beispiel 45.1 greifen zwei Threads auf die int-Variable **a** zu, um sie zu inkrementieren. Anstatt auf einen Lock zuzugreifen, wird `boost::atomic` verwendet, um den Zugriff auf **a** zu synchronisieren. Wenn Sie das Beispielprogramm ausführen, wird `2` ausgegeben.
Der Trick hinter `boost::atomic` ist, dass es Prozessoren gibt, die atomare Zugriffe auf Variablen unterstützen. Ist die Inkrementierung einer int-Variablen eine atomare Operation, ist ein Lock nicht notwendig. Wird das Beispielprogramm auf einer Plattform ausgeführt, die keine atomare Inkrementierung einer int-Variablen kennt, verwendet `boost::atomic` einen Lock.

Beispiel 45.2 `boost::atomic` mit oder ohne Lock

```
#include <boost/atomic.hpp>
#include <iostream>

int main()
{
  std::cout.setf(std::ios::boolalpha);

  boost::atomic<short> s;
  std::cout << s.is_lock_free() << '\n';

  boost::atomic<int> i;
  std::cout << i.is_lock_free() << '\n';

  boost::atomic<long> l;
  std::cout << l.is_lock_free() << '\n';
}
```

Sie können für eine atomare Variable `is_lock_free()` aufrufen, um zu überprüfen, ob der Zugriff auf diese Variable ohne Lock stattfindet. Wenn Sie das Beispielprogramm auf einem Intel x86-Prozessor ausführen, wird dreimal `true` ausgegeben. Führen Sie es auf einem Prozessor aus, der keinen lock-freien Zugriff auf short-, int- und long-Variablen bietet, wird `false` ausgegeben.
Boost.Atomic bietet mit `BOOST_ATOMIC_INT_LOCK_FREE` oder `BOOST_ATOMIC_LONG_LOCK_FREE` Makros an, um auch zur Kompilierung festzustellen, welche Typen einen lock-freien Zugriff unterstützen.
Beachten Sie, dass im Beispiel 45.2 ausschließlich integrale Typen verwendet werden. Sie dürfen `boost::atomic` nicht mit Klassen wie `std::string` oder `std::vector` verwenden. Boost.Atomic unterstützt integrale Typen, triviale Klassen, Zeiger und bool. Beispiele für integrale Typen sind short, int oder long. Triviale Klassen definieren Objekte, die mit `std::memcpy()` kopiert werden können.

Beispiel 45.3 `boost::atomic` mit **`boost::memory_order_seq_cst`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_seq_cst);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Beispiel 45.3 inkrementiert **a** zweimal – diesmal nicht mit `operator++`, sondern über den Aufruf der Methode `fetch_add()`. Diese Methode erwartet als Parameter nicht nur eine Zahl, um die **a** erhöht werden soll. Es muss außerdem eine *Memory-Order* übergeben werden.
Die Memory-Order bestimmt, in welcher Reihenfolge Zugriffe auf den Speicher erfolgen müssen. Diese Reihenfolge ist standardmäßig nicht festgelegt und ergibt sich nicht aus der Reihenfolge von Code-Zeilen. So dürfen Compiler und Prozessor Reihenfolgen beliebig ändern, solange sich ein Programm von außen betrachtet so verhält als wären Speicherzugriffe in der Reihenfolge erfolgt, wie sie im Quellcode niedergeschrieben sind. Diese Regel gilt jedoch ausschließlich pro Thread. Wird mehr als ein Thread verwendet, kann eine geänderte Reihenfolge von Speicherzugriffen dazu führen, dass sich ein Programm fehlerhaft verhält. Boost.Atomic ermöglicht es, beim Zugriff auf Variablen eine Memory-Order anzugeben, um sicherzustellen, dass Speicherzugriffe in einem Multithreaded-Programm in der gewünschten Reihenfolge erfolgen.

> **Anmerkung**
>
> Beachten Sie, dass Boost.Atomic nicht nur eine höhere Performance ermöglicht, indem genau die Memory-Order angegeben werden können, die benötigt werden. Memory-Order erhöhen den Schwierigkeitsgrad beträchtlich. Code wird durch Memory-Order wesentlich komplizierter.

Im Beispiel 45.3 wird die Memory-Order **`boost::memory_order_seq_cst`** verwendet, um **a** um 1 zu erhöhen. Die Memory-Order steht für *sequentielle Konsistenz* – auf Englisch sequential consistency. Dies ist die restriktivste Memory-Order. Sie bedeutet, dass alle Speicherzugriffe, die im Code vor dem Aufruf von `fetch_add()` angegeben sind, auch vor dem Aufruf dieser Methode erfolgen müssen. Alle Speicherzugriffe, die im Code nach dem Aufruf von `fetch_add()` angegeben sind, müssen nach dem Aufruf dieser Methode erfolgen. Compiler und Prozessor dürfen Speicherzugriffe vor und nach dem Aufruf von `fetch_add()` beliebig ordnen. Sie dürfen aber zum Beispiel keinen Speicherzugriff, der vor dem Aufruf von `fetch_add()` im Code angegeben ist, nach dem Aufruf dieser Methode vornehmen. **`boost::memory_order_seq_cst`** ist eine scharfe Grenze, die nach oben und unten gilt.

`boost::memory_order_seq_cst` ist die restriktivste Memory-Order. Sie wird standardmäßig verwendet, wenn Sie auf `boost::atomic`-Variablen zugreifen und keine Memory-Order explizit angeben. Die Inkrementierung von **a** mit `operator++` im Beispiel 45.1 erfolgte ebenfalls über **`boost::memory_order_seq_cst`**.

`boost::memory_order_seq_cst` ist nicht immer notwendig. So gibt es im Beispiel 45.3 keinen Grund, Speicherzugriffe auf andere Variablen zu synchronisieren – es gibt schließlich keine anderen Variablen, auf die beide Threads zugreifen oder die von **a** abhängen. **a** wird in `main()` auf die Standardausgabe ausgegeben. Dies erfolgt jedoch, nachdem beide Threads beendet wurden. Der Aufruf von `join()` garantiert, dass der Wert in **a** erst dann gelesen wird, wenn die Threads beendet wurden.

Beispiel 45.4 `boost::atomic` mit **`memory_order_relaxed`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_relaxed);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Im Beispiel 45.4 wurde die Memory-Order in **`boost::memory_order_relaxed`** geändert. Dies ist die am wenigsten restriktive Memory-Order: Sie erlaubt eine beliebige Neuordnung der Speicherzugriffe. Das Beispielprogramm funktioniert auch mit dieser Memory-Order wie gewünscht, weil es außer auf die Variable **a** keine anderen Speicherzugriffe in den Threads gibt. Es ist keine bestimmte Reihenfolge von Speicherzugriffen nötig.

Beispiel 45.5 `boost::atomic` mit **`memory_order_release`** und **`memory_order_acquire`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};
int b = 0;

void thread1()
```

```
{
  b = 1;
  a.store(1, boost::memory_order_release);
}

void thread2()
{
  while (a.load(boost::memory_order_acquire) != 1)
    ;
  std::cout << b << '\n';
}

int main()
{
  std::thread t1{thread1};
  std::thread t2{thread2};
  t1.join();
  t2.join();
}
```

Zwischen der restriktivsten Memory-Order **`boost::memory_order_seq_cst`** und der am wenigsten restriktiven **`boost::memory_order_relaxed`** liegen mehrere Abstufungen. Im Beispiel 45.5 werden die Memory-Order **`boost::memory_order_release`** und **`boost::memory_order_acquire`** vorgestellt.
`boost::memory_order_release` stellt sicher, dass Speicherzugriffe, die im Code vor **`boost::memory_order_release`** stattfinden, auch tatsächlich vorher geschehen. Compiler und Prozessor dürfen Speicherzugriffe von vor **`boost::memory_order_release`** nicht danach ausführen. Sie dürfen jedoch Speicherzugriffe, die im Code hinter **`boost::memory_order_release`** stehen, vorziehen.
`boost::memory_order_acquire` funktioniert genauso wie **`boost::memory_order_release`**, bezieht sich aber auf Speicherzugriffe nach **`boost::memory_order_acquire`**. Während **`boost::memory_order_release`** eine Schranke für vorhergehende Speicherzugriffe ist, schränkt **`boost::memory_order_acquire`** nachfolgende Speicherzugriffe ein. Diese dürfen bei **`boost::memory_order_acquire`** nicht vorgezogen werden, während vorhergehende Speicherzugriffe durchaus nachgezogen werden dürfen.
Im Beispiel 45.5 wird im ersten Thread **`boost::memory_order_release`** eingesetzt, um sicherzustellen, dass **b** auf 1 gesetzt ist, bevor **a** auf 1 gesetzt wird. **`boost::memory_order_release`** garantiert, dass der Speicherzugriff auf **b** nicht nach dem Speicherzugriff auf **a** erfolgt.
Um beim Zugriff auf **a** eine Memory-Order angeben zu können, wird die Methode `store()` aufgerufen. Sie entspricht einer Zuweisung mit `operator=`.
Im zweiten Thread wird in einer Schleife **a** gelesen. Dies erfolgt über die Methode `load()`. Hier wird ebenfalls nicht auf den Zuweisungsoperator zugegriffen, damit eine Memory-Order angeben werden kann. Dies ist hier **`boost::memory_order_acquire`**.
`boost::memory_order_acquire` stellt im zweiten Thread sicher, dass der Speicherzugriff auf **b** nicht vor dem Zugriff auf **a** erfolgt. Der zweite Thread wartet in der Schleife darauf, dass **a** vom ersten Thread auf 1 gesetzt wird. Ist dies geschehen, wird **b** ausgegeben.
Wenn Sie das Beispielprogramm ausführen, wird `1` auf die Standardausgabe ausgegeben. Die verwendeten Memory-Order stellen sicher, dass die Speicherzugriffe in der richtigen Reihenfolge erfolgen. Der erste Thread schreibt immer zuerst 1 in **b**, bevor der zweite Thread auf **b** zugreift und den Wert ausliest.

> **Tipp**
>
> Weiterführende Informationen rund um atomare Variablen finden Sie zum Beispiel in einem Artikel über Memory Order im GCC-Wiki.

Kapitel 46

Boost.Lockfree

Boost.Lockfree bietet thread-safe und lock-freie Container an. Sie können auf die von dieser Bibliothek bereitgestellten Container von mehreren Threads aus zugreifen, ohne Zugriffe synchronisieren zu müssen.
In der Version 1.57.0 bietet Boost.Lockfree lediglich zwei Container an: Eine Queue vom Typ `boost::lockfree::queue` und einen Stack vom Typ `boost::lockfree::stack`. Für die Queue wird eine zweite Implementation in Form der Klasse `boost::lockfree::spsc_queue` angeboten, die für Anwendungsfälle optimiert ist, in denen genau ein Thread in die Queue schreibt und genau ein Thread aus der Queue liest. Die Abkürzung spsc im Klassennamen steht für single producer/single consumer.

Beispiel 46.1 `boost::lockfree::spsc_queue` in Aktion

```
#include <boost/lockfree/spsc_queue.hpp>
#include <thread>
#include <iostream>

boost::lockfree::spsc_queue<int> q{100};
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  consume();
  std::cout << sum << '\n';
}
```

Beispiel 46.1 verwendet den Container `boost::lockfree::spsc_queue`. Im ersten Thread, der die Funktion `produce()` ausführt, werden dem Container die Zahlen 1 bis 100 hinzugefügt. Der zweite Thread, der `consume()` ausführt, liest die Zahlen aus dem Container aus und addiert sie zu **`sum`**. Da der Container `boost::lockfree::spsc_queue` explizit einen derartigen gleichzeitigen Zugriff von zwei Threads aus unterstützt, ist es nicht notwendig, die Threads zu synchronisieren.
Beachten Sie, dass die Funktion `consume()` ein zweites Mal aufgerufen wird, nachdem die Threads beendet wurden. Dies ist notwendig, damit zwingend alle 100 Zahlen addiert werden und die Summe 5050 errechnet

wird. Da `consume()` in einer Schleife auf die Queue zugreift, könnte es sein, dass Zahlen schneller ausgelesen werden als sie mit `produce()` der Queue hinzugefügt werden. Ist die Queue leer, gibt `pop()` `false` zurück. Der Thread, der `consume()` ausführt, könnte also beendet werden, weil `produce()` im anderen Thread die Queue nicht schnell genug füllen konnte. Wenn der Thread, der `produce()` ausführt, beendet wird, ist jedoch klar, dass alle Zahlen der Queue hinzugefügt wurden. Der zweite Aufruf von `consume()` stellt sicher, dass möglicherweise noch nicht ausgelesene Zahlen zu **`sum`** addiert werden.
Beachten Sie, dass die Größe der Queue dem Konstruktor als Parameter übergeben wird. Da `boost::lockfree::spsc_queue` als Ringspeicher implementiert ist, hat die Queue im Beispiel 46.1 eine Kapazität von 100 Elementen. Kann ein Element der Queue nicht hinzugefügt werden, weil sie voll ist, gibt `push()` `false` zurück. Im Beispielprogramm wird der Rückgabewert von `push()` nicht überprüft, weil der Queue genau 100 Zahlen hinzugefügt werden. Eine Kapazität von 100 ist demnach ausreichend.

Beispiel 46.2 `boost::lockfree::spsc_queue` mit `boost::lockfree::capacity`

```
#include <boost/lockfree/spsc_queue.hpp>
#include <boost/lockfree/policies.hpp>
#include <thread>
#include <iostream>

using namespace boost::lockfree;

spsc_queue<int, capacity<100>> q;
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  while (q.consume_one([](int i){ sum += i; }))
    ;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  q.consume_all([](int i){ sum += i; });
  std::cout << sum << '\n';
}
```

Beispiel 46.2 funktioniert wie das vorherige. Diesmal wird die Größe des Ringbuffers jedoch zur Kompilierung angegeben. Dazu wird auf die Template-Klasse `boost::lockfree::capacity` zugegriffen und die Kapazität als Template-Parameter übergeben. **`q`** wird daraufhin über den Standardkonstruktor instanziiert – die Kapazität kann nicht mehr zur Laufzeit angegeben werden.
Die Funktion `consume()` wurde dahingehend geändert, dass Zahlen nicht mehr über `pop()` gelesen werden. Stattdessen wird `consume_one()` aufgerufen und als Parameter eine Lambda-Funktion übergeben. `consume_one()` liest ebenso wie `pop()` eine Zahl aus. Die Zahl wird jedoch nicht über eine Referenz an den Aufrufer übergeben. Sie wird als einziger Parameter an die Lambda-Funktion weitergereicht.
In `main()` wird, wenn die Threads beendet wurden, für die Queue statt der Funktion `consume()` die Methode `consume_all()` aufgerufen. `consume_all()` funktioniert wie `consume_one()`, stellt aber sicher, dass die Queue nach dem Methodenaufruf leer ist. Mit `consume_all()` wird die Lambda-Funktion also genauso oft aufgerufen wie Elemente in der Queue sind.
Wenn Sie Beispiel 46.2 ausführen, wird wieder `5050` ausgegeben.

Beispiel 46.3 `boost::lockfree::queue` mit variabler Container-Größe

```
#include <boost/lockfree/queue.hpp>
#include <thread>
#include <atomic>
```

```
#include <iostream>

boost::lockfree::queue<int> q{100};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

Beispiel 46.3 führt `consume()` in zwei Threads aus. Da mehr als ein Thread lesend auf die Queue zugreift, darf nicht mehr der Typ `boost::lockfree::spsc_queue` verwendet werden. Im Beispielprogramm kommt `boost::lockfree::queue` zum Einsatz.
Zugriffe auf die Variable **`sum`** sind nun außerdem dank `std::atomic` thread-safe.
Die Größe der Queue ist auf 100 gesetzt – dies ist der Parameter, der an den Konstruktor übergeben wird. Dabei handelt es sich jedoch lediglich um die Anfangsgröße. `boost::lockfree::queue` ist standardmäßig nicht als Ringspeicher implementiert. Werden der Queue mehr Elemente hinzugefügt als die Kapazität erlaubt, wird die Kapazität automatisch erweitert. `boost::lockfree::queue` reserviert also dynamisch zusätzlichen Speicherplatz, wenn die Anfangsgröße nicht ausreicht.
Dies bedeutet, dass `boost::lockfree::queue` nicht zwingend lock-frei ist. Der Allokator, der standardmäßig von `boost::lockfree::queue` verwendet wird, ist `boost::lockfree::allocator`. `boost::lockfree::allocator` wiederum basiert auf `std::allocator`. Es hängt also von diesem Allokator ab, ob `boost::lockfree::queue` ohne Einschränkungen lock-frei ist.

Beispiel 46.4 `boost::lockfree::queue` mit konstanter Container-Größe

```
#include <boost/lockfree/queue.hpp>
#include <thread>
#include <atomic>
#include <iostream>

using namespace boost::lockfree;

queue<int, fixed_sized<true>> q{10000};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
```

```
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

Im Beispiel 46.4 wird die Queue auf eine feste Größe von 10000 Elementen gesetzt. Die Queue reserviert nicht mehr zusätzlich Speicher, wenn sie voll ist. 10000 ist eine feste Obergrenze.
Die Kapazität der Queue ist konstant, weil `boost::lockfree::fixed_sized` als Template-Parameter übergeben wird. Die Kapazität wird als Parameter an den Konstruktor übergeben und kann jederzeit über die Methode `reserve()` neu gesetzt werden. Möchten Sie die Kapazität zur Kompilierung angeben, können Sie `boost::lockfree::queue` auch den Template-Parameter `boost::lockfree::capacity` übergeben. `boost::lockfree::capacity` schließt `boost::lockfree::fixed_sized` ein.
Die Queue im Beispiel 46.4 hat eine Kapazität von 10000 Elementen. Da `consume()` 10000 Zahlen der Queue hinzufügt, wird die Obergrenze nicht überschritten. Würde die Obergrenze überschritten werden können, würde `push()` `false` zurückgeben.
`boost::lockfree::queue` ähnelt `boost::lockfree::spsc_queue` und bietet ebenfalls Methoden wie `consume_one()` und `consume_all()` an.
Die dritte Klasse `boost::lockfree::stack`, die von Boost.Lockfree angeboten wird, ist den anderen beiden sehr ähnlich. Wie bei `boost::lockfree::queue` können `boost::lockfree::fixed_size` und `boost::lockfree::capacity` als Template-Parameter angegeben werden. Die Methoden unterscheiden sich ebenfalls kaum.

Kapitel 47

Boost.MPI

Boost.MPI bietet eine Schnittstelle zum MPI-Standard (Message Passing Interface) an. Dieser Standard vereinfacht die Entwicklung von Programmen, die Aufgaben parallel lösen sollen. Sie können derartige Programme auch entwickeln, indem Sie auf Threads zugreifen oder sie so strukturieren, dass mehrere Prozesse über Shared Memory oder Netzwerkverbindungen miteinander kommunizieren. Der Vorteil von MPI ist, dass Sie sich um diese Details nicht kümmern müssen. Sie können sich ganz auf die Parallelisierung Ihres Programms konzentrieren.
Ein Nachteil von MPI ist, dass MPI-Programme eine entsprechende Laufzeitumgebung voraussetzen. MPI bietet sich nur dann an, wenn Sie die Laufzeitumgebung kontrollieren können. Möchten Sie zum Beispiel ein Programm verteilen, das wie gewohnt per Doppelklick gestartet werden können soll, müssen Sie auf MPI verzichten. Während Threads, Shared Memory und Netzwerke von Betriebssystemen von Haus aus unterstützt werden, bieten Betriebssysteme üblicherweise keine Laufzeitumgebung für MPI-Programme an. Anwender müssten zusätzliche Schritte unternehmen, um Ihr MPI-Programm ausführen zu können.

47.1 Entwicklungs- und Laufzeitumgebung

MPI ist ein Standard, der zahlreiche Funktionen zum *Parallel Computing* definiert. Damit sind Programme gemeint, die so strukturiert sind, dass sie Aufgaben gleichzeitig ausführen können, weil sie in Laufzeitumgebungen eingesetzt werden, die das gleichzeitige Ausführen von Aufgaben ermöglichen. Derartige Laufzeitumgebungen basieren üblicherweise auf mehreren Prozessoren. Da ein Prozessor Codeanweisungen nur nacheinander ausführen kann, entsteht erst durch den Zusammenschluss mehrerer Prozessoren eine Laufzeitumgebung, die eine parallele Codeausführung ermöglicht. Werden mehrere tausend Prozessoren zusammengeschaltet, erhält man Parallelrechner – Architekturen, wie sie in Supercomputern vorgefunden werden. MPI stammt aus genau diesem Umfeld, weil man nach Wegen suchte, Supercomputer einfacher programmieren zu können.
Wenn Sie MPI verwenden möchten, brauchen Sie eine Implementation des Standards. So definiert MPI zwar zahlreiche Funktionen. Das heißt aber nicht, dass Ihr Betriebssystem diese Funktionen von Haus aus zur Verfügung stellt. So bieten zum Beispiel die Desktop-Versionen von Windows keine MPI-Funktionen an.
Die beiden wichtigsten MPI-Implementationen sind MPICH und Open MPI. MPICH ist eine der ersten MPI-Implementationen und existiert seit Mitte der 90er Jahre. Es handelt sich um eine ausgereifte und portable Implementation, die bis heute gepflegt und aktualisiert wird. Die erste Version von Open MPI wurde 2005 veröffentlicht. Da Open MPI jedoch in Zusammenarbeit zahlreicher Entwickler entsteht, die für frühere MPI-Implementationen verantwortlich zeichnen, gilt Open MPI als neuer aufkommender Standard. Das heißt nicht, dass MPICH abgeschrieben werden darf. Es gibt zahlreiche MPI-Implementationen, die auf MPICH basieren. Microsoft liefert zum Beispiel mit dem Microsoft HPC Pack seine eigene auf MPICH basierende Implementation aus.
MPICH stellt für zahlreiche Betriebssysteme wie Windows, Linux und OS X Installationsdateien zur Verfügung. Wenn Sie eine MPI-Implementation benötigen und sie diese nicht selbst von Quellcode kompilieren möchten, sind die MPICH-Installationsdateien der schnellste Weg, um in MPI einzusteigen.
Die MPICH-Installationsdateien stellen Ihnen die notwendigen Headerdateien und Bibliotheken zur Verfügung, um MPI-Programme entwickeln zu können. Darüberhinaus erhalten Sie die für MPI-Programme notwendige Laufzeitumgebung. Da MPI-Programme Aufgaben parallel auf mehreren Prozessoren ausführen, laufen sie in mehreren Prozessen ab. Ein MPI-Programm wird nicht einmal gestartet, sondern mehrfach. So laufen zeitgleich mehrere Instanzen eines MPI-Programms auf mehreren Prozessoren, die über Funktionen miteinander kommunizieren, die der MPI-Standard definiert.

Ein MPI-Programm können Sie daher nicht wie gewohnt per Doppelklick starten. Sie verwenden ein Hilfsprogramm, das üblicherweise mpiexec heißt. Diesem Hilfsprogramm übergeben Sie Ihr MPI-Programm, das dann in der MPI-Laufzeitumgebung ausgeführt wird. Über zusätzliche Parameter können Sie angeben, wie viele Prozesse für Ihr MPI-Programm gestartet werden sollen und wie sie miteinander kommunizieren sollen – also zum Beispiel über Sockets oder Shared Memory. Da sich die MPI-Laufzeitumgebung um diese Details kümmert, können Sie sich auf die parallele Programmierung konzentrieren.
Wenn Sie sich für die Installationsdateien von MPICH entschieden haben: Beachten Sie, dass MPICH ausschließlich eine 64-Bit-Version zur Verfügung stellt. Sie müssen einen 64-Bit-Compiler verwenden, um MPI-Programme mit MPICH entwickeln zu können. Sämtliche Boost-Bibliotheken einschließlich Boost.MPI müssen ebenfalls als 64-Bit-Version vorliegen.

47.2 Einfacher Datenaustausch

Boost.MPI ist eine C++-Schnittstelle zum MPI-Standard. Die Bibliothek stellt Klassen und Funktionen bereit, die alle im Namensraum `boost::mpi` definiert sind. Sie müssen lediglich die Headerdatei `boost/mpi.hpp` einbinden, um auf alle Klassen und Funktionen der Bibliothek Zugriff zu erhalten.

Beispiel 47.1 MPI-Umgebung und Kommunikator

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::cout << world.rank() << ", " << world.size() << '\n';
}
```

Beispiel 47.1 zeigt Ihnen ein erstes einfaches MPI-Programm. In diesem Programm werden zwei Klassen verwendet, die Sie in allen folgenden Beispielprogrammen immer wieder antreffen werden. `boost::mpi::environment` verwenden Sie, um MPI zu initialisieren. So ruft der Konstruktor dieser Klasse die Funktion `MPI_Init()` aus dem MPI-Standard auf, der Destruktor `MPI_Finalize()`. `boost::mpi::communicator` wiederum verwenden Sie, um einen *Kommunikator* zu erstellen. Der Kommunikator ist eines der zentralen Konzepte von MPI und ermöglicht einen einfachen Datenaustausch zwischen Prozessen.
`boost::mpi::environment` ist eine sehr einfache Klasse, die nur wenige Methoden anbietet. So können Sie zum Beispiel über `initialized()` herausfinden, ob MPI erfolgreich initialisiert wurde. Die Methode gibt ein Ergebnis vom Typ bool zurück. `processor_name()` gibt den Namen des aktuellen Prozesses als `std::string` zurück. Und über `abort()` können Sie ein MPI-Programm stoppen – also nicht nur den aktuellen Prozess. Sie übergeben dieser Methode als einzigen Parameter einen int-Wert, der der Rückgabewert des Programms an die MPI-Laufzeitumgebung darstellt. Für die meisten MPI-Programme werden Sie diese Methoden aber nicht benötigen. Üblicherweise instanziieren Sie `boost::mpi::environment` zum Programmanfang und greifen danach nicht mehr auf die Instanz zu – so wie im Beispiel 47.1 und allen folgenden Beispielprogrammen in diesem Kapitel.
Wesentlich interessanter ist die Klasse `boost::mpi::communicator`. Es handelt sich hierbei um einen Kommunikator, der die Verbindung zu anderen Prozessen herstellt, aus denen das MPI-Programm besteht. Jedem Prozess ist ein Rang zugeordnet. Dabei handelt es sich um eine Ganzzahl – alle Prozesse sind also nummeriert. Ein Prozess kann seinen Rang erfahren, indem er `rank()` für den Kommunikator aufruft. Möchte ein Prozess wissen, aus wie vielen Prozessen ein MPI-Programm besteht, kann er `size()` aufrufen.
Wenn Sie Beispiel 47.1 ausführen wollen, müssen Sie dies mit einem Hilfsprogramm machen, das von der von Ihnen verwendeten MPI-Implementation angeboten wird. Bei MPICH heißt dieses Hilfsprogramm mpiexec. Starten Sie Beispiel 47.1 zum Beispiel mit folgendem Befehl:

```
mpiexec -n 4 beispiel.exe
```

mpiexec erwartet neben dem Namen des MPI-Programms, das es starten soll, eine Angabe zur Anzahl der Prozesse. So teilen Sie mpiexec zum Beispiel über `-n 4` mit, dass vier Prozesse gestartet werden sollen. Ihr Programm wird daraufhin viermal ausgeführt. Dabei handelt es sich jedoch nicht um gänzlich unabhängige Prozesse. Die Prozesse stehen über die MPI-Laufzeitumgebung in Kontakt. Sie gehören alle dem gleichen Kommunikator an. Jedem Prozess ist in diesem Kommunikator ein Rang zugeordnet. Starten Sie Beispiel 47.1 mit vier Prozessen, werden für `rank()` die Zahlen 0 bis 3 zurückgegeben und für `size()` 4.

Beachten Sie, dass die Ausgabe des Programms durcheinander sein kann. Es handelt sich schließlich um vier Prozesse, die gleichzeitig auf die Standardausgabe zugreifen. Ob zum Beispiel der Prozess mit Rang 0 oder der mit Rang 3 zuerst Daten auf die Standardausgabe ausgibt, lässt sich nicht vorhersagen. Es ist außerdem möglich, dass ein Prozess einen anderen beim Schreiben auf die Standardausgabe unterbricht und ein Prozess seinen Rang nicht direkt gefolgt von der Größe des Kommunikators ausgeben kann.

Beispiel 47.2 Blockierende Funktionen zum Datenversand und -empfang

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    int i;
    world.recv(1, 16, i);
    std::cout << i << '\n';
  }
  else if (world.rank() == 1)
  {
    world.send(0, 16, 99);
  }
}
```

`boost::mpi::communicator` bietet mit `send()` und `recv()` zwei einfache Methoden an, um Daten zwischen zwei Prozessen auszutauschen. Es handelt sich hierbei um blockierende Methoden, die erst zurückkehren, wenn Daten gesendet oder empfangen wurden. Das ist vor allem beim Aufruf von `recv()` wichtig. Wenn Sie `recv()` aufrufen, ohne dass ein anderer Prozess Daten sendet, kann der Methodenaufruf nicht zurückkehren und der entsprechende Prozess würde im Methodenaufruf verharren.
Beispiel 47.2 ist so strukturiert, dass der Prozess mit Rang 0 mit `recv()` Daten empfängt, während der Prozess mit Rang 1 mit `send()` Daten sendet. Starten Sie das Beispiel mit mehr als zwei Prozessen, werden alle anderen Prozesse beendet, ohne dass sie etwas tun.
Sie übergeben drei Parameter an `send()`: Der erste Parameter ist der Rang des Prozesses, an den Daten gesendet werden sollen. Der zweite Parameter ist ein *Tag*, der Daten identifiziert. Der dritte Parameter sind die Daten selbst, die gesendet werden sollen.
Der Tag ist immer eine Ganzzahl. Im Beispiel 47.2 ist dies die Zahl 16. Der Tag ermöglicht es, den Aufruf von `send()` zu identifizieren. So werden Sie gleich sehen, dass der Tag auch beim Aufruf von `recv()` zum Einsatz kommt.
Als dritten Parameter wird im Beispiel 47.2 die Zahl 99 an `send()` übergeben. Diese Zahl soll vom Prozess mit dem Rang 1 an den Prozess mit dem Rang 0 gesendet werden. Boost.MPI unterstützt alle primitiven Typen aus C++, so dass Sie einen int-Wert wie 99 problemlos senden können.
`recv()` erwartet ähnliche Parameter wie `send()`. Der erste Parameter ist der Rang des Prozesses, von dem Daten empfangen werden sollen. Der zweite Parameter ist der Tag, der den Aufruf von `recv()` mit dem entsprechenden Aufruf von `send()` verbindet. Der dritte Parameter ist eine Variable, in der die empfangenen Daten abgelegt werden sollen.
Wenn Sie Beispiel 47.2 mit mindestens zwei Prozessen ausführen, wird `99` auf die Standardausgabe ausgegeben.

Beispiel 47.3 Daten von einem beliebigen Prozess empfangen

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    int i;
    world.recv(boost::mpi::any_source, 16, i);
    std::cout << i << '\n';
```

```
  }
  else
  {
    world.send(0, 16, world.rank());
  }
}
```

Beispiel 47.3 basiert auf dem vorherigen. Jetzt wird nicht mehr die Zahl 99 gesendet, sondern der Rang des Prozesses, der `send()` aufruft. `send()` wird außerdem nicht mehr nur vom Prozess mit dem Rang 1 aufgerufen, sondern von allen Prozessen, die einen höheren Rang als 0 haben.
Der Aufruf von `recv()` ist ebenfalls geändert worden. Als erster Parameter wird **`boost::mpi::any_source`** übergeben. Damit wartet `recv()` nicht mehr auf Daten, die von einem bestimmten Prozess gesendet werden. Egal, welcher Prozess den Tag 16 sendet – die Daten werden vom Prozess mit dem Rang 0 empfangen.
Wenn Sie Beispiel 47.3 mit genau zwei Prozessen ausführen, wird `1` auf die Standardausgabe ausgegeben. Es gibt in diesem Fall nur einen Prozess, der `send()` aufruft – der Prozess mit dem Rang 1. Starten Sie das Programm mit mehr als zwei Prozessen, lässt sich die Ausgabe nicht vorhersagen. Schließlich rufen in diesem Fall mehrere Prozesse `send()` auf und versuchen, ihren Rang zu senden. Es hängt vom Zufall ab, welcher Rang auf die Standardausgabe ausgegeben wird.

Beispiel 47.4 Sender mit `boost::mpi::status` ermitteln

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    int i;
    boost::mpi::status s = world.recv(boost::mpi::any_source, 16, i);
    std::cout << s.source() << ": " << i << '\n';
  }
  else
  {
    world.send(0, 16, 99);
  }
}
```

`recv()` besitzt einen Rückgabewert vom Typ `boost::mpi::status`. Diese Klasse bietet unter anderem eine Methode `source()` an, die den Rang des Prozesses zurückgibt, von dem Daten empfangen wurden. Wenn Sie Beispiel 47.4 ausführen, erfahren Sie, von welchem Prozess die Zahl 99 empfangen wurde.
Bisher wurden `send()` und `recv()` verwendet, um eine int-Zahl zu übertragen. Im Beispiel 47.5 soll eine Zeichenkette zwischen zwei Prozessen ausgetauscht werden.

Beispiel 47.5 Ein Array mit `send()` und `recv()` übertragen

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    char buffer[14];
    world.recv(boost::mpi::any_source, 16, buffer, 13);
    buffer[13] = '\0';
    std::cout << buffer << '\n';
  }
  else
  {
    const char *c = "Hello, world!";
```

```
    world.send(0, 16, c, 13);
  }
}
```

`send()` und `recv()` können nicht nur einzelne Werte übertragen, sondern auch Arrays. So wird im Beispiel 47.5 eine Zeichenkette gesendet, die in einem char-Array gespeichert ist. Da `send()` und `recv()` primitive Typen wie char unterstützen, kann ein char-Array ohne Probleme übertragen werden.
Beim Aufruf von `send()` wird als dritter Parameter ein Zeiger auf die Zeichenkette und als vierter Parameter die Länge der Zeichenkette übergeben. `recv()` erhält als dritten Parameter einen Zeiger auf ein Array, in dem die empfangenen Daten gespeichert werden sollen. Als vierten Parameter wird `recv()` die Anzahl der Zeichen übergeben, die empfangen und im Array **`buffer`** abgelegt werden sollen. Wenn Sie Beispiel 47.5 ausführen, wird `Hello, world!` auf die Standardausgabe ausgegeben.

Beispiel 47.6 Einen String mit `send()` und `recv()` übertragen

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    std::string s;
    world.recv(boost::mpi::any_source, 16, s);
    std::cout << s << '\n';
  }
  else
  {
    std::string s = "Hello, world!";
    world.send(0, 16, s);
  }
}
```

Auch wenn Boost.MPI nur primitive Typen unterstützt, heißt das nicht, dass Sie keine Objekte nicht-primitiver Typen senden und empfangen können. Boost.MPI arbeitet mit Boost.Serialization zusammen. Objekte, die nach den Regeln von Boost.Serialization serialisiert werden können, können von Boost.MPI übertragen werden.
Im Beispiel 47.6 wird „Hello, world!“ übertragen. Diesmal handelt es sich nicht um eine Zeichenkette in einem char-Array, sondern um einen String. Da dieser den Typ `std::string` hat, kann er nur übertragen werden, wenn er serialisiert werden kann. Boost.Serialization bietet die Headerdatei `boost/serialization/string.hpp` an, die Sie lediglich einbinden müssen, damit Strings serialisierbar werden.
Wenn Sie Objekte vom Typ benutzerdefinierter Klassen übertragen möchten, sehen Sie sich im Kapitel 64 an, wie Sie im Detail vorgehen müssen.

47.3 Asynchroner Datenaustausch

Boost.MPI unterstützt neben den blockierenden Methoden `send()` und `recv()` auch einen asynchronen Datenaustausch. Sie greifen hierzu auf die Methoden `isend()` und `irecv()` zu. Die Namen beginnen mit einem i, um darauf hinzuweisen, dass diese Methoden sofort – auf Englisch immediate – zurückkehren.

Beispiel 47.7 Asynchroner Datenempfang mit `irecv()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
```

```
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    std::string s;
    boost::mpi::request r = world.irecv(boost::mpi::any_source, 16, s);
    if (r.test())
      std::cout << s << '\n';
    else
      r.cancel();
  }
  else
  {
    std::string s = "Hello, world!";
    world.send(0, 16, s);
  }
}
```

Beispiel 47.7 verwendet die bereits bekannte blockierende Methode `send()`, um den String „Hello, world!“ zu senden. Der Datenempfang findet mit der asynchronen Methode `irecv()` statt. Diese Methode erwartet die gleichen Parameter wie `recv()`. Der entscheidende Unterschied ist: Wenn `irecv()` zurückkehrt, ist nicht garantiert, dass in der Variablen **s** Daten empfangen wurden.
`irecv()` gibt ein Objekt vom Typ `boost::mpi::request` zurück. Sie können für dieses Objekt `test()` aufrufen, um festzustellen, ob Daten empfangen wurden oder nicht. Diese Methode gibt ein Ergebnis vom Typ bool zurück. Sie können `test()` beliebig oft aufrufen. Da es sich bei `irecv()` um eine asynchrone Methode handelt, kann es sein, dass bei einem ersten Aufruf zum Beispiel `false`, beim zweiten Aufruf jedoch `true` zurückgegeben wird. In einem derartigen Fall wäre die asynchrone Operation nach dem ersten Aufruf, aber vor dem zweiten abgeschlossen worden.
Im Beispiel 47.7 wird `test()` nur einmal aufgerufen. Wurden Daten in **s** empfangen, wird die Variable auf die Standardausgabe ausgegeben. Wurden keine Daten empfangen, wird die asynchrone Operation abgebrochen. Dies geschieht über den Aufruf von `cancel()`.
Wenn Sie Beispiel 47.7 mehrfach ausführen, erhalten Sie manchmal die Ausgabe `Hello, world!` und manchmal keine Ausgabe. Ob eine Ausgabe erfolgt, hängt davon ab, ob die Daten vor dem Aufruf von `test()` empfangen werden konnten.

Beispiel 47.8 Auf mehrere asynchrone Operationen mit `wait_all()` warten

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    boost::mpi::request requests[2];
    std::string s[2];
    requests[0] = world.irecv(1, 16, s[0]);
    requests[1] = world.irecv(2, 16, s[1]);
    boost::mpi::wait_all(requests, requests + 2);
    std::cout << s[0] << "; " << s[1] << '\n';
  }
  else if (world.rank() == 1)
  {
    std::string s = "Hello, world!";
    world.send(0, 16, s);
  }
  else if (world.rank() == 2)
  {
    std::string s = "Hello, moon!";
    world.send(0, 16, s);
  }
```

```
}
```

Sie können für ein Objekt vom Typ `boost::mpi::request` mehrfach `test()` aufrufen, um festzustellen, wann die asynchrone Operation abgeschlossen wurde. Sie können jedoch auch wie im Beispiel 47.8 eine blockierende Funktion wie `boost::mpi::wait_all()` aufrufen. Auch wenn es sich hierbei um eine blockierende Funktion handelt: Der Vorteil von `boost::mpi::wait_all()` ist, dass diese Funktion auf den Abschluss mehrerer asynchroner Operationen warten kann. Im Falle von `boost::mpi::wait_all()` müssen alle asynchronen Operationen abgeschlossen sein, bevor `boost::mpi::wait_all()` zurückkehrt.
Im Beispiel 47.8 sendet der Prozess mit Rang 1 „Hello, world!“ und der Prozess mit Rang 2 „Hello, moon!“. Da weder klar ist, in welcher Reihenfolge die Daten im Prozess mit Rang 0 empfangen werden, noch dass die Reihenfolge eine Rolle spielen würde, werden die Daten mit `irecv()` empfangen. Da die Datenausgabe jedoch erst dann erfolgen kann, wenn alle asynchronen Operationen abgeschlossen und alle Daten empfangen wurden, werden die Rückgabeobjekte vom Typ `boost::mpi::request` an `boost::mpi::wait_all()` übergeben. `boost::mpi::wait_all()` erwartet zwei Iteratoren, die auf den Anfang und das Ende eines Containers zeigen. Da es sich im Beispiel 47.8 um ein Array handelt, werden ein Zeiger auf den Anfang und ein Zeiger auf das Ende des Arrays übergeben.
Boost.MPI bietet neben `boost::mpi::wait_all()` weitere blockierende Funktionen an, um auf den Abschluss mehrerer asynchroner Operationen zu warten. `boost::mpi::wait_any()` kehrt zurück, wenn genau eine asynchrone Operation abgeschlossen wurde. `boost::mpi::wait_some()` kehrt zurück, wenn eine oder mehrere asynchrone Operationen abgeschlossen wurden. Beide Funktionen geben ein Ergebnis vom Typ `std::pair` zurück, das Auskunft darüber gibt, welche asynchrone Operation oder Operationen abgeschlossen wurden. Neben Funktionen, um auf den Abschluss asynchroner Operationen zu warten, bietet Boost.MPI mit `boost::mpi::test_all()`, `boost::mpi::test_any()` und `boost::mpi::test_some()` auch eine Möglichkeit, den Status mehrerer asynchroner Operationen mit einem einzigen Funktionsaufruf zu überprüfen. Diese Funktionen sind nicht-blockierend und kehren sofort zurück.

47.4 Kollektiver Datenaustausch

Den in diesem Kapitel bisher vorgestellten Funktionen ist gemein, dass es eine Eins-zu-eins-Beziehung zwischen Prozessen gibt: Es gibt einen Prozess, der sendet, und einen Prozess, der empfängt. Die Verknüpfung erfolgt über den Tag. In diesem Abschnitt lernen Sie Funktionen kennen, die mit den gleichen Parametern für mehrere Prozesse aufgerufen werden können und für den einen Prozess Datenversand und für den anderen Prozess Datenempfang bedeuten. Diese Funktionen werden als *kollektive Operationen* bezeichnet.

Beispiel 47.9 Daten von mehreren Prozessen mit `gather()` empfangen

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    std::vector<std::string> v;
    boost::mpi::gather<std::string>(world, "", v, 0);
    std::ostream_iterator<std::string> out{std::cout, "\n"};
    std::copy(v.begin(), v.end(), out);
  }
  else if (world.rank() == 1)
  {
    boost::mpi::gather(world, std::string{"Hello, world!"}, 0);
  }
  else if (world.rank() == 2)
  {
```

```
    boost::mpi::gather(world, std::string{"Hello, moon!"}, 0);
  }
}
```

Im Beispiel 47.9 wird für mehrere Prozesse die Funktion `boost::mpi::gather()` aufgerufen. Ob diese Funktion Daten sendet oder empfängt, hängt von den Parametern ab, die ihr übergeben werden.
Die Prozesse mit den Rängen 1 und 2 nutzen `boost::mpi::gather()` zum Datenversand. Dazu übergeben sie die zu sendenden Daten – hier die Strings „Hello, world!“ und „Hello, moon!“ – und den Rang des Prozesses, an den die Daten gesendet werden sollen. Da es sich bei `boost::mpi::gather()` um eine Funktion und keine Methode handelt, muss außerdem der Kommunikator **`world`** übergeben werden.
Der Prozess mit Rang 0 ruft `boost::mpi::gather()` zum Datenempfang auf. Da die empfangenen Daten irgendwo gespeichert werden müssen, wird mit **`v`** ein Objekt vom Typ std::vector<std::string> übergeben. Beachten Sie, dass Sie für `boost::mpi::gather()` diesen Typ verwenden müssen. Es werden keine anderen Container oder String-Typen unterstützt.
Es müssen im Prozess mit Rang 0 außerdem die gleichen Parameter übergeben werden, die von den Prozessen mit Rang 1 und 2 verwendet werden. So wird auch im Prozess mit Rang 0 **`world`**, ein zu sendender String und 0 an `boost::mpi::gather()` übergeben.
Wenn Sie Beispiel 47.9 ausführen, wird `Hello, world!` und `Hello, moon!` in zwei Zeilen auf die Standardausgabe ausgegeben. Wenn Sie sich die Ausgabe genau ansehen, stellen Sie fest, dass jedoch zuerst eine leere Zeile ausgegeben wird. In dieser ersten Zeile wird der leere String ausgegeben, der vom Prozess mit Rang 0 an `boost::mpi::gather()` übergeben wird. Im Vektor **`v`** befinden sich also drei Strings, die von den Prozessen mit den Rängen 0, 1 und 2 empfangen wurden. Die Daten werden dabei gemäß den Rängen der Prozesse abgelegt. Die Ränge der Prozesse werden als Index verwendet, um die Daten im Vektor zu speichern. Wenn Sie das Beispiel mehrfach ausführen, erhalten Sie immer an erster Stelle im Vektor den leeren String, an zweiter Stelle den String „Hello, world!“ und an den dritter Stelle den String „Hello, moon!“.
Beachten Sie, dass Sie Beispiel 47.9 nicht mit mehr als drei Prozessen ausführen dürfen. Wenn Sie beim Aufruf von mpiexec zum Beispiel `-n 4` angeben, gibt das Programm keine Daten aus. Es bleibt außerdem hängen und muss mit CTRL+C abgebrochen werden.
Kollektive Operationen müssen immer für alle Prozesse ausgeführt werden. Wenn Ihr Programm eine Funktion wie `boost::mpi::gather()` aufruft, müssen Sie sicherstellen, dass der Aufruf für alle Prozesse erfolgt. Sie verletzen sonst den MPI-Standard. Da eine Funktion wie `boost::mpi::gather()` für alle Prozesse ausgerufen werden muss, erfolgt der Aufruf üblicherweise nicht wie im Beispiel 47.9 mehrfach für unterschiedliche Prozesse. Sehen Sie sich dazu Beispiel 47.10 an, das grundsätzlich genauso funktioniert wie Beispiel 47.9.

Beispiel 47.10 Mit `gather()` Daten von allen Prozessen sammeln

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 1)
    s = "Hello, world!";
  else if (world.rank() == 2)
    s = "Hello, moon!";
  std::vector<std::string> v;
  boost::mpi::gather(world, s, v, 0);
  std::ostream_iterator<std::string> out{std::cout, "\n"};
  std::copy(v.begin(), v.end(), out);
}
```

Sie rufen Funktionen, die kollektive Operationen anstoßen, üblicherweise für alle Prozesse auf. Diese Funktionen sind dabei so definiert, dass klar ist, welche Aufgaben die Prozesse zu erledigen haben – obwohl alle Prozesse die gleichen Parameter beim Aufruf übergeben.

Im Beispiel 47.10 wird `boost::mpi::gather()` verwendet. Diese Funktion – der Name deutet es an – sammelt Daten. Die Daten werden dabei in dem Prozess gesammelt, dessen Rang als letzter Parameter an `boost::mpi::gather()` übergeben wird. Dieser Prozess sammelt Daten, die er von allen Prozessen erhält. Der Vektor, in dem die Daten gespeichert werden sollen, wird ausschließlich von dem Prozess verwendet, der die Daten sammeln soll.
Beachten Sie, dass `boost::mpi::gather()` Daten von allen Prozessen sammelt. Dies schließt den Prozess mit ein, in dem die Daten gesammelt werden sollen. Im Beispiel 47.10 ist dies der Prozess mit Rang 0. Dieser Prozess sendet beim Aufruf von Beispiel 47.10 einen leeren String in der Variablen **s** an sich selbst. Der leere String wird im Vektor **v** abgelegt. Wie Sie anhand der folgenden Beispiele sehen werden, schließen kollektive Operationen immer alle Prozesse ein.
Sie können Beispiel 47.10 mit beliebig vielen Prozessen ausführen, da für jeden Prozess ein Aufruf von `boost::mpi::gather()` stattfindet. Wenn Sie das Beispiel mit drei Prozessen starten, erhalten Sie die gleiche Datenausgabe wie beim vorherigen Beispiel.

Beispiel 47.11 Mit `scatter()` Daten an alle Prozesse verteilen

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <vector>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::vector<std::string> v{"Hello, world!", "Hello, moon!",
    "Hello, sun!"};
  std::string s;
  boost::mpi::scatter(world, v, s, 0);
  std::cout << world.rank() << ": " << s << '\n';
}
```

Beispiel 47.11 stellt die Funktion `boost::mpi::scatter()` vor. Es handelt sich dabei um das Gegenstück zu `boost::mpi::gather()`. Während `boost::mpi::gather()` Daten von mehreren Prozessen in einem Prozess sammelt, verteilt `boost::mpi::scatter()` Daten von einem Prozess an mehrere Prozesse.
Im Beispiel 47.11 werden die Daten im Vektor **v** vom Prozess mit Rang 0 an alle Prozesse verteilt. Dies schließt den eigenen Prozess – also den mit Rang 0 – ein. Der String „Hello, world!“ wird vom Prozess mit Rang 0 in der Variablen **s** empfangen. Der Prozess mit Rang 1 empfängt „Hello, moon!“ in dieser Variablen. Der Prozess mit Rang 2 erhält „Hello, sun!“ in **s**.

Beispiel 47.12 Mit `broadcast()` Daten an alle Prozesse senden

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  boost::mpi::broadcast(world, s, 0);
  std::cout << s << '\n';
}
```

`boost::mpi::broadcast()` sendet Daten von einem Prozess an alle Prozesse. Der Unterschied zu `boost::mpi::scatter()` ist, dass die gleichen Daten an alle Prozesse gesendet werden. So erhalten im Beispiel 47.12 alle Prozesse den String „Hello, world!“ in der Variablen **s**. Alle Prozesse geben `Hello, world!` auf die Standardausgabe aus.

Beispiel 47.13 Mit `reduce()` Daten sammeln und auswerten

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

std::string min(const std::string &lhs, const std::string &rhs)
{
  return lhs.size() < rhs.size() ? lhs : rhs;
}

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  else if (world.rank() == 1)
    s = "Hello, moon!";
  else if (world.rank() == 2)
    s = "Hello, sun!";
  std::string result;
  boost::mpi::reduce(world, s, result, min, 0);
  if (world.rank() == 0)
    std::cout << result << '\n';
}
```

`boost::mpi::reduce()` sammelt so wie `boost::mpi::gather()` Daten von mehreren Prozessen. Diese werden jedoch nicht in einem Vektor abgelegt. `boost::mpi::reduce()` erwartet, dass eine Funktion oder ein Funktionsobjekt übergeben wird, das zur Auswertung der Daten verwendet wird.
Im Beispiel 47.13 erhält der Prozess mit Rang 0 als Ergebnis in der Variablen **`result`** den String „Hello, sun!“. Der Aufruf von `boost::mpi::reduce()` führt dazu, dass die Strings, die die verschiedenen Prozesse in **`s`** an die Funktion übergeben, gesammelt und ausgewertet werden. Die Auswertung findet mit Hilfe der Funktion `min()` statt, die `boost::mpi::reduce()` als vierter Parameter übergeben wird. Diese Funktion vergleicht jeweils zwei Strings und gibt den kürzeren der beiden als Ergebnis zurück.
Wenn Sie Beispiel 47.13 mit drei Prozessen ausführen, wird `Hello, sun!` auf die Standardausgabe ausgegeben. Starten Sie mehr als drei Prozesse, wird eine leere Zeile ausgegeben, da alle Prozesse mit einem Rang größer als 2 in der Variablen **`s`** einen leeren String an `boost::mpi::reduce()` übergeben, der kürzer ist als alle anderen Strings.

Beispiel 47.14 Mit `all_reduce()` Daten sammeln und auswerten

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

std::string min(const std::string &lhs, const std::string &rhs)
{
  return lhs.size() < rhs.size() ? lhs : rhs;
}

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  else if (world.rank() == 1)
    s = "Hello, moon!";
  else if (world.rank() == 2)
```

```
    s = "Hello, sun!";
  std::string result;
  boost::mpi::all_reduce(world, s, result, min);
  std::cout << world.rank() << ": " << result << '\n';
}
```

Im Beispiel 47.14 kommt die Funktion `boost::mpi::all_reduce()` zum Einsatz. Diese Funktion sammelt und wertet wie `boost::mpi::reduce()` Daten aus. Der Unterschied zwischen den beiden Funktionen ist, dass `boost::mpi::all_reduce()` das Ergebnis an alle Prozesse verteilt, während `boost::mpi::reduce()` das Ergebnis nur an den Prozess weitergibt, dessen Rang als letzter Parameter übergeben wurde. Wie Sie anhand des Funktionsaufrufs im Beispiel 47.14 sehen, wird an `boost::mpi::all_reduce()` kein Rang übergeben.
Wenn Sie Beispiel 47.14 mit drei Prozessen ausführen, gibt jeder Prozess `Hello, sun!` auf die Standardausgabe aus.

47.5 Kommunikatoren

Alle bisherigen Beispiele haben einen einzigen Kommunikator verwendet, in dem alle Prozesse vereint waren. Es ist jedoch möglich, mehrere Kommunikatoren zu erstellen, um eine bestimmte Anzahl von Prozessen zusammenzufassen. Das ist vor allem im Zusammenhang mit kollektiven Operationen nützlich. Denn die entsprechenden Funktionen müssen nicht für alle Prozesse aufgerufen werden. Sie müssen lediglich für alle Prozesse in dem Kommunikator aufgerufen werden, der als erster Parameter an die entsprechenden Funktionen übergeben wird.

Beispiel 47.15 Mit mehreren Kommunikatoren arbeiten

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  boost::mpi::communicator local = world.split(world.rank() < 2 ? 99 : 100);
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  boost::mpi::broadcast(local, s, 0);
  std::cout << world.rank() << ": " << s << '\n';
}
```

Im Beispiel 47.15 kommt die bereits bekannte Funktion `boost::mpi::broadcast()` zum Einsatz. Die Funktion sendet den String „Hello, world!“ vom Prozess mit Rang 0 an alle Prozesse, die im Kommunikator **`local`** vereint sind. Der Prozess mit Rang 0 muss dabei diesem Kommunikator angehören.
Der Kommunikator **`local`** wird über den Aufruf von `split()` erstellt. Es handelt sich dabei um eine Methode, die für den globalen Kommunikator **`world`** aufgerufen wird. `split()` erwartet eine Zahl vom Typ int, um Prozesse in Kommunikatoren einzuordnen. Prozesse, die die gleiche Zahl übergeben, landen im gleichen Kommunikator. Welche Zahl genau an `split()` übergeben wird, spielt keinen Rolle. Es kommt lediglich darauf an, welche Prozesse die gleichen Zahlen übergeben.
Im Beispiel 47.15 übergeben die ersten beiden Prozesse mit den Rängen 0 und 1 die Zahl 99 an `split()`. Wird das Programm mit mehr als zwei Prozessen gestartet, übergeben diese die Zahl 100. Das bedeutet, dass es einen lokalen Kommunikator gibt, in dem sich die ersten beiden Prozesse befinden, und einen anderen lokalen Kommunikator, in dem sich alle anderen Prozesse befinden. Jeder Prozess gehört dem Kommunikator an, der von `split()` zurückgegeben wird. Inwiefern sich in diesem Kommunikator andere Prozesse befinden, hängt davon ab, ob andere Prozesse die gleiche Zahl an `split()` übergeben.
Beachten Sie, dass sich Ränge immer auf einen Kommunikator beziehen. Dabei beginnt die Zählweise immer bei 0. So hat im Beispiel 47.15 der Prozess, der bezogen auf den globalen Kommunikator den Rang 0 hat, bezogen auf den neuen lokalen Kommunikator ebenfalls den Rang 0. Der Prozess, der bezogen auf den globalen Kommunikator den Rang 2 hat, hat bezogen auf den neuen lokalen Kommunikator auch den Rang 0. Es gibt zwei lokale Kommunikatoren, nachdem zwei Prozesse die Zahl 99 und alle andere Prozesse die Zahl 100 an `split()` übergeben.

Wenn Sie Beispiel 47.15 mit zwei oder mehr Prozessen ausführen, wird zweimal `Hello, world!` ausgegeben – und zwar von den Prozessen mit den Rängen 0 und 1 bezogen auf den globalen Kommunikator. Da nur für den Prozess mit Rang 0 die Variable **`s`** auf „Hello, world!“ gesetzt wird, wird dieser String über den lokalen Kommunikator nur an die Prozesse gesendet, die sich mit dem Prozess mit Rang 0 in der gleichen Gruppe befinden. Dies ist ausschließlich der Prozess mit Rang 1, da nur dieser die gleiche Zahl an `split()` übergibt wie der Prozess mit Rang 0.

Beispiel 47.16 Prozesse mit `group` gruppieren

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <boost/range/irange.hpp>
#include <boost/optional.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  boost::mpi::group local = world.group();
  boost::integer_range<int> r = boost::irange(0, 1);
  boost::mpi::group subgroup = local.exclude(r.begin(), r.end());
  boost::mpi::communicator others{world, subgroup};
  std::string s;
  boost::optional<int> rank = subgroup.rank();
  if (rank)
  {
    if (rank == 0)
      s = "Hello, world!";
    boost::mpi::broadcast(others, s, 0);
  }
  std::cout << world.rank() << ": " << s << '\n';
}
```

MPI ermöglicht, Prozessgruppen explizit zusammenzustellen. Dazu wird die Klasse `boost::mpi::group` angeboten. Wenn Sie die Methode `group()` für einen Kommunikator aufrufen, erhalten Sie alle Prozesse des Kommunikators in einem Objekt vom Typ `boost::mpi::group`. Sie können dieses Objekt nicht zur Kommunikation verwenden. Die Klasse `boost::mpi::group` dient ausschließlich dazu, eine Gruppierung zu verändern, um einen neuen Kommunikator zu erstellen.

`boost::mpi::group` bietet Methoden wie `include()` und `exclude()` an. Sie übergeben diesen Methoden Iteratoren, um Prozesse in der Gruppe ein- oder auszuschließen. `include()` und `exclude()` geben daraufhin eine neue Gruppe vom Typ `boost::mpi::group` zurück.

Im Beispiel 47.16 werden `exclude()` zwei Iteratoren übergeben, die auf ein Objekt vom Typ `boost::integer_range` zeigen. Dieses Objekt repräsentiert einen Wertebereich bestehend aus Ganzzahlen. Das Objekt wird mit Hilfe der Funktion `boost::irange()` erstellt, die eine Unter- und Obergrenze für den Wertebereich erwartet. Wichtig ist, dass der zweite Parameter die Zahl ist, die gerade nicht mehr zum Wertebereich zählt. Das bedeutet, dass im Wertebereich der Variablen **`r`** ausschließlich die Zahl 0 enthalten ist.

Der Aufruf von `exclude()` führt dazu, dass eine neue Gruppe **`subgroup`** entsteht, die alle Prozesse außer den mit Rang 0 enthält. Diese Gruppe wird verwendet, um einen neuen Kommunikator **`others`** zu erstellen. Dazu muss sowohl der globale Kommunikator **`world`** als auch die Gruppe **`subgroup`** an den Konstruktor von `boost::mpi::communicator` übergeben werden.

Beachten Sie, dass mit **`others`** ein Kommunikator erstellt wurde, der für den Prozess mit Rang 0 leer ist. Da dieser Prozess dem Kommunikator nicht angehört, die Variable **`others`** im Prozess mit Rang 0 aber natürlich trotzdem existiert, müssen Sie darauf achten, dass Sie den Kommunikator in diesem Prozess nicht verwenden. Im Beispiel 47.16 geschieht dies über den Aufruf von `rank()`. Diese Methode gibt für Prozesse, die der Gruppe nicht angehören, ein leeres Objekt vom Typ `boost::optional` zurück. Alle anderen Prozesse erhalten ihren Rang bezogen auf die Gruppe.

Wenn `rank()` einen Rang und kein leeres Objekt vom Typ `boost::optional` zurückgibt, wird `boost::mpi::broadcast()` aufgerufen. Der Prozess mit Rang 0 sendet den String „Hello, world!“ an alle Prozesse im Kommunikator **`others`**. Beachten Sie, dass sich der Rang hierbei auf den ersten Prozess in diesem Kommunikator bezieht. Dies ist der Prozess, der bezogen auf den globalen Kommunikator **`world`** den Rang 1 hat.

Wenn Sie Beispiel 47.16 mit mehr als zwei Prozessen ausführen, wird für alle Prozesse mit einem Rang größer als 0 `Hello, world!` ausgegeben.

Teil XI

Generische Programmierung

Die Bibliotheken Boost.TypeTraits, Boost.EnableIf und Boost.Fusion unterstützen die generische Programmierung. Sie können sie auch ohne tiefergehende Kenntnisse zur Template Meta-Programmierung einsetzen.

- Boost.TypeTraits bietet Funktionen, um Typen auf ihre Eigenschaften zu überprüfen.
- Boost.EnableIf können Sie zusammen mit Boost.TypeTraits einsetzen, um zum Beispiel Funktionen anhand des Typs ihres Rückgabewerts zu überladen.
- Mit Boost.Fusion ist es möglich, heterogene Container zu erstellen – also Container, die Elemente mit unterschiedlichen Typen speichern können.

Kapitel 48

Boost.TypeTraits

Typen haben unterschiedliche Eigenschaften, die sich die generische Programmierung zunutze macht. Dazu ist es notwendig, Typen auf Eigenschaften zu überprüfen oder ihre Eigenschaften zu ändern. Boost.TypeTraits ist die Bibliothek, die die dazu erforderlichen Hilfsmittel bietet.
Seit C++11 befindet sich ein Teil der Funktionen, die Boost.TypeTraits anbietet, in der Standardbibliothek. Sie können auf diese Funktionen über die Headerdatei `type_traits` zugreifen. Boost.TypeTraits bietet jedoch bedeutend mehr Funktionen an als Sie in `type_traits` vorfinden.

Beispiel 48.1 Typ-Kategorien abfragen

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_integral<int>::value << '\n';
  std::cout << is_floating_point<int>::value << '\n';
  std::cout << is_arithmetic<int>::value << '\n';
  std::cout << is_reference<int>::value << '\n';
}
```

Im Beispiel 48.1 kommen verschiedene Funktionen zum Einsatz, mit denen Typ-Kategorien abgefragt werden können. So wird mit `boost::is_integral` überprüft, ob ein Typ integral ist – ob er also Ganzzahlen speichern kann. Mit `boost::is_floating_point` wird überprüft, ob ein Typ Kommazahlen speichern kann. `boost::is_arithmetic` wird verwendet, um zu überprüfen, ob ein Typ arithmetische Operationen unterstützt. Und `boost::is_reference` kann verwendet werden, um herauszufinden, ob ein Typ eine Referenz ist.
Beachten Sie, dass sich `boost::is_integral` und `boost::is_floating_point` gegenseitig ausschließen. Ein Typ speichert entweder Ganz- oder Kommazahlen. `boost::is_arithmetic` und `boost::is_reference` sind hingegen kategorieübergreifend. So werden zum Beispiel arithmetische Operationen sowohl von Ganz- als auch Kommazahlen unterstützt.
Alle Funktionen von Boost.TypeTraits bieten mit **`value`** ein Ergebnis an, das entweder `true` oder `false` ist. Wenn Sie Beispiel 48.1 ausführen, wird für `is_integral<int>` und `is_arithmetic<int>` `true` und für `is_floating_point<int>` und `is_reference<int>` `false` ausgegeben. Da es sich bei den Funktionen um Templates handelt, findet keine Berechnung zur Laufzeit statt. Das Beispielprogramm verhält sich als wäre `true` und `false` in den verschiedenen Zeilen direkt angegeben worden.
Im Beispiel 48.1 war das Ergebnis der verschiedenen Funktionen ein bool-Wert. Dieser konnte direkt auf die Standardausgabe ausgegeben werden. Soll ein Ergebnis von einer Template-Funktion weiterverarbeitet werden, darf es nicht als bool-Wert vorliegen, sondern muss ein Typ sein.

Beispiel 48.2 `boost::true_type` und `boost::false_type`

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;
```

```
int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_same<is_integral<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_floating_point<int>::type, false_type>::value <<
    '\n';
  std::cout << is_same<is_arithmetic<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_reference<int>::type, false_type>::value << '\n';
}
```

Neben **value** bieten die Funktionen von Boost.TypeTraits das Ergebnis auch in type an. Während **value** ein bool-Wert ist, ist type ein Typ. So wie es bei **value** nur zwei Werte gibt – `true` oder `false` – gibt es auch bei type nur zwei Ergebnistypen: `boost::true_type` oder `boost::false_type`. Dank type ist es möglich, das Ergebnis einer Funktion als Typ an eine andere Funktion zu übergeben.
Im Beispiel 48.2 wird mit `boost::is_same` eine weitere Funktion von Boost.TypeTraits verwendet. Diese Funktion erwartet zwei Typen als Parameter und überprüft, ob es sich dabei um die gleichen Typen handelt. Um die Ergebnisse von `boost::is_integral`, `boost::is_floating_point`, `boost::is_arithmetic` und `boost::is_reference` an `boost::is_same` übergeben zu können, muss auf type zugegriffen werden. type wird daraufhin mit `boost::true_type` oder `boost::false_type` verglichen. Die Ergebnisse von `boost::is_same` werden wiederum über **value** entgegengenommen. Weil es sich dabei um einen bool-Wert handelt, kann er auf die Standardausgabe ausgegeben werden.

Beispiel 48.3 Mit Boost.TypeTraits Typ-Eigenschaften abfragen

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << has_plus<int>::value << '\n';
  std::cout << has_pre_increment<int>::value << '\n';
  std::cout << has_trivial_copy<int>::value << '\n';
  std::cout << has_virtual_destructor<int>::value << '\n';
}
```

Beispiel 48.3 stellt einige Funktionen vor, mit denen Eigenschaften von Typen abgefragt werden können. `boost::has_plus` überprüft, ob ein Typ den Operator `operator+` unterstützt und sich somit zwei Objekte des gleichen Typs verknüpfen lassen. `boost::has_pre_increment` gibt an, ob ein Typ den Prä-Inkrement-Operator `operator++` unterstützt. Mit `boost::has_trivial_copy` kann ermittelt werden, ob ein Typ einen trivialen Copy-Konstruktor hat. Und mit `boost::has_virtual_destructor` kann überprüft werden, ob ein Typ einen virtuellen Destructor hat.
Wenn Sie Beispiel 48.3 ausführen, wird dreimal `true` und einmal `false` ausgegeben.

Beispiel 48.4 Mit Boost.TypeTraits Typ-Eigenschaften ändern

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_const<add_const<int>::type>::value << '\n';
  std::cout << is_same<remove_pointer<int*>::type, int>::value << '\n';
  std::cout << is_same<make_unsigned<int>::type, unsigned int>::value <<
    '\n';
  std::cout << is_same<add_rvalue_reference<int>::type, int&&>::value <<
    '\n';
}
```

Beispiel 48.4 zeigt Ihnen, wie Sie Typ-Eigenschaften ändern. So wird einem Typ zum Beispiel mit `boost::add_const` ein `const` hinzugefügt. Ist ein Typ bereits konstant, ändert sich nichts. Der Code kompiliert problemlos, und der Typ bleibt konstant.
`boost::remove_pointer` entfernt das Sternchen von einem Zeiger und gibt den Typ zurück, auf den der Zeiger zeigt. `boost::make_unsigned` macht aus einem Typ einen ohne Vorzeichen. Und `boost::add_rvalue_reference` wandelt einen Typ in eine Rvalue-Referenz um.
Wenn Sie Beispiel 48.4 ausführen, wird viermal `true` ausgegeben.

Kapitel 49

Boost.EnableIf

Boost.EnableIf ermöglicht es, überladene Template-Funktionen oder spezialisierte Template-Klassen zu deaktivieren. Die Deaktivierung führt dazu, dass die entsprechenden Templates vom Compiler nicht berücksichtigt werden. Dies hilft nicht nur, Situationen zu vermeiden, in denen der Compiler nicht weiß, welche überladene Template-Funktion er heranziehen soll. Es macht es auch einfach, Templates zu definieren, die nicht nur für einen bestimmten Typ, sondern für eine Gruppe von Typen gelten sollen.
Boost.EnableIf ist seit C++11 Teil der Standardbibliothek. Sie können auf die in diesem Kapitel vorgestellten Funktionen ohne Boost-Bibliothek zugreifen, wenn Sie die Headerdatei `type_traits` einbinden.

Beispiel 49.1 Mit `boost::enable_if` Funktionen über ihren Rückgabewert überladen

```
#include <boost/utility/enable_if.hpp>
#include <type_traits>
#include <string>
#include <iostream>

template <typename T>
typename boost::enable_if<std::is_same<T, int>, T>::type create()
{
  return 1;
}

template <typename T>
typename boost::enable_if<std::is_same<T, std::string>, T>::type create()
{
  return "Boost";
}

int main()
{
  std::cout << create<std::string>() << '\n';
}
```

Beispiel 49.1 definiert eine Template-Funktion `create()`, die ein Objekt von dem Typ zurückgeben soll, der als Template-Parameter übergeben wird. Das Objekt wird innerhalb von `create()` initialisiert, so dass kein Parameter an `create()` übergeben werden muss und kann. Die Parameterlisten der beiden `create()`-Funktionen unterscheiden sich nicht. So gesehen handelt es sich nicht um eine überladene Funktion. Der Compiler würde einen Fehler melden, wenn nicht mit Hilfe von Boost.EnableIf eine der beiden Template-Funktionen aktiviert und die andere deaktiviert würde.
Boost.EnableIf stellt die Klasse `boost::enable_if` zur Verfügung. Es handelt sich um ein Template, das zwei Parameter erwartet. Der erste Parameter gibt eine Bedingung an. Der zweite Parameter ist der Typ des gesamten `boost::enable_if`-Ausdrucks, wenn die Bedingung wahr ist. Der Trick ist, dass dieser Typ nicht existiert, wenn die Bedingung falsch ist. Der `boost::enable_if`-Ausdruck ergibt in diesem Fall ungültigen Code. Bei Templates beschwert sich der Compiler jedoch nicht, wenn er auf ungültigen Code trifft. Stattdessen ignoriert er das entsprechende Template und sucht nach einem anderen, das passen könnte. Dieses Konzept ist auch als SFINAE bekannt, was für „Substitution Failure Is Not An Error“ steht.
Im Beispiel 49.1 greifen beide Bedingungen im `boost::enable_if`-Ausdruck auf die Klasse `std::is_same` zu. Diese Klasse stammt aus der C++11-Standardbibliothek und ermöglicht den Vergleich zweier Typen. Da ein

derartiger Vergleich entweder wahr oder falsch ist, reicht es aus, `std::is_same` zur Definition einer Bedingung zu verwenden.
Ist eine Bedingung wahr, soll die entsprechende `create()`-Funktion ein Objekt von dem Typ zurückgeben, der beim Aufruf von `create()` als Template-Parameter angegeben ist. Daher wird als zweiter Parameter an `boost::enable_if` der Template-Parameter T übergeben. Der gesamte `boost::enable_if`-Ausdruck wird durch T ersetzt, wenn eine Bedingung wahr ist. Der Compiler sieht im Beispiel 49.1 also entweder eine Funktion, die ein int zurückgibt, oder eine Funktion, die ein `std::string` zurückgibt. Sollte `create()` mit einem anderen Typ als int oder `std::string` aufgerufen werden, würde der Compiler einen Fehler melden.
Wenn Sie Beispiel 49.1 ausführen, wird `Boost` ausgegeben.

Beispiel 49.2 Mit `boost::enable_if` Funktionen für eine Gruppe von Typen spezialisieren

```
#include <boost/utility/enable_if.hpp>
#include <type_traits>
#include <iostream>

template <typename T>
void print(typename boost::enable_if<std::is_integral<T>, T>::type i)
{
  std::cout << "Integral: " << i << '\n';
}

template <typename T>
void print(typename boost::enable_if<std::is_floating_point<T>, T>::type f)
{
  std::cout << "Floating point: " << f << '\n';
}

int main()
{
  print<short>(1);
  print<long>(2);
  print<double>(3.14);
}
```

Beispiel 49.2 verwendet `boost::enable_if`, um eine Funktion für eine Gruppe von Typen zu spezialisieren. Die entsprechende Funktion heißt `print()` und erwartet einen Parameter. Sie könnte daher problemlos überladen werden. Beim Überladen muss jedoch ein konkreter Typ angegeben werden. Wenn die Funktion für eine Gruppe von Typen wie short, int und long das Gleiche tun soll, kann mit `boost::enable_if` eine entsprechende Bedingung definiert werden. Im Beispiel 49.2 wird hierzu auf `std::is_integral` zugegriffen. Die zweite `print()`-Funktion wird mit `std::is_floating_point` für alle Kommazahlen überladen – also für float und double.

Kapitel 50

Boost.Fusion

Die Standardbibliothek bietet zahlreiche Container an, die eines gemeinsam haben: Sie sind homogen. In Containern der Standardbibliothek können nur Werte eines Typs gespeichert werden. So können in einem Vektor vom Typ std::vector<int> nur int-Zahlen gespeichert werden, und in einem Vektor vom Typ std::vector<std::string> nur Strings.
Boost.Fusion ermöglicht es, heterogene Container zu erstellen. So können Sie zum Beispiel einen Vektor erstellen, dessen erstes Element eine int-Zahl und dessen zweites Element ein String ist. Boost.Fusion unterstützt aber nicht nur heterogene Container. Die Bibliothek stellt auch Algorithmen zur Verfügung, um diese zu verarbeiten. Stellen Sie sich Boost.Fusion als die Standardbibliothek für heterogene Container vor.
Genaugenommen existiert mit `std::tuple` seit C++11 auch in der Standardbibliothek ein heterogener Container. Wenn Sie ein Tuple definieren, können Sie unterschiedliche Typen für die Werte angeben, die Sie im Tuple speichern möchten. Boost.Fusion bietet mit `boost:fusion::tuple` einen ähnlichen Typ an. Während die Standardbibliothek rund um Tuple nicht viel zu bieten hat, sind Tuple für Boost.Fusion lediglich ein Ausgangsbaustein.

Beispiel 50.1 Fusion-Tuple verarbeiten

```
#include <boost/fusion/tuple.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  std::cout << get<0>(t) << '\n';
  std::cout << get<1>(t) << '\n';
  std::cout << std::boolalpha << get<2>(t) << '\n';
  std::cout << get<3>(t) << '\n';
}
```

Im Beispiel 50.1 wird ein Tuple bestehend aus einem int, `std::string`, bool und double definiert. Das Tuple basiert dabei nicht auf dem aus der Standardbibliothek bekannten Typ `std::tuple`, sondern auf `boost:fusion::tuple`. Im nächsten Schritt wird das Tuple instanziiert und mit verschiedenen Werten initialisiert. Abschließend wird mit `boost::fusion::get()` auf die Elemente des Tuples zugegriffen, um sie auf die Standardausgabe auszugeben. `boost::fusion::get()` ist gleichbedeutend mit der Funktion `std::get()`, mit der auf Elemente in einem `std::tuple` zugegriffen werden kann.
Fusion-Tuple unterscheiden sich nicht von Tuplen aus der Standardbibliothek. So ist es zum Beispiel auch wenig verwunderlich, dass Boost.Fusion eine Funktion `boost::fusion::make_tuple()` anbietet, die genauso funktioniert wie `std::make_tuple()`. Folgende Beispiele stellen aufbauend auf Fusion-Tuplen jedoch Möglichkeiten vor, die die Standardbibliothek nicht bietet.

Beispiel 50.2 Mit `boost::fusion::for_each()` über einen Tuple iterieren

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/algorithm.hpp>
#include <string>
```

```
#include <iostream>

using namespace boost::fusion;

struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  for_each(t, print{});
}
```

Im Beispiel 50.2 lernen Sie den Algorithmus `boost::fusion::for_each()` kennen, mit dem über Fusion-Container iteriert werden kann. Die Funktion wird hier eingesetzt, um die Werte im Tuple **`t`** nacheinander auf die Standardausgabe auszugeben.
`boost::fusion::for_each()` ist dem Algorithmus `std::for_each()` nachempfunden. Während `std::for_each()` ausschließlich über homogene Container iterieren kann, steht mit `boost::fusion::for_each()` ein Algorithmus für heterogene Container zur Verfügung.
Beachten Sie, dass Sie keine Iteratoren, sondern einen Container direkt als ersten Parameter an `boost::fusion::for_each()` übergeben. Möchten Sie nicht über alle Elemente in einem Container iterieren, können Sie mit Hilfe einer *View* einen Teilausschnitt erstellen.

Beispiel 50.3 Mit `boost::fusion::filter_view` einen Fusion-Container filtern

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/view.hpp>
#include <boost/fusion/algorithm.hpp>
#include <boost/type_traits.hpp>
#include <boost/mpl/arg.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  filter_view<tuple_type, boost::is_integral<boost::mpl::arg<1>>> v{t};
  for_each(v, print{});
}
```

Boost.Fusion bietet Views an, die sich wie Container verhalten, im Gegensatz zu Containern jedoch keine Daten speichern. Mit Views kann auf Daten in einem Container anders zugegriffen werden als es der entsprechende Container ermöglicht. Insofern ähneln Views Adaptern aus Boost.Range. Während Adapter jedoch ausschließlich auf einen Container angewandt werden können, können Views auch Daten aus mehreren Containern abbilden.

Im Beispiel 50.3 wird die Klasse `boost::fusion::filter_view` verwendet, um das Tuple **`t`** zu filtern. Es sollen mit `boost::fusion::for_each()` nicht mehr alle Elemente auf die Standardausgabe geschrieben werden, sondern nur noch die, die auf einem integralen Typ basieren.
`boost::fusion::filter_view` erwartet als ersten Template-Parameter den Typ des Containers, der gefiltert werden soll. Als zweiter Template-Parameter muss ein Prädikat übergeben werden, das die Elemente filtert. Das Prädikat muss dabei die Elemente basierend auf ihrem Typ filtern.
Boost.Fusion heißt so, weil es zwei Welten vereint: C++-Programme arbeiten mit Werten zur Laufzeit und mit Typen während der Kompilierung. Für Entwickler stehen üblicherweise Werte zur Laufzeit im Vordergrund. Auch sind die meisten Bestandteile der Standardbibliothek darauf ausgerichtet, Werte zur Laufzeit einfacher verarbeiten zu können. Sollen Typen während der Kompilierung verarbeitet werden, kommt die Template Meta-Programmierung ins Spiel. Während zur Laufzeit Werte in Abhängigkeit von anderen Werten verarbeitet werden, werden zur Kompilierung Typen in Abhängigkeit von anderen Typen verarbeitet. Boost.Fusion macht es möglich, Werte in Abhängigkeit von Typen zu verarbeiten.
Der zweite an `boost::fusion::filter_view` übergebene Template-Parameter ist ein Prädikat, das auf jeden Typ im Tuple angewandt wird. Das Prädikat erhält als Parameter seinerseits einen Typ und muss als Ergebnis `true` zurückgeben, wenn der entsprechende Typ in der View enthalten sein soll. Wird `false` zurückgegeben, wird der entsprechende Typ gefiltert.
Im Beispiel 50.3 wird auf die Klasse `boost::is_integral` aus Boost.TypeTraits zugegriffen. Diesem Template muss als Parameter ein Typ aus dem Tuple übergeben werden, um zu überprüfen, ob er integral ist oder nicht. Da diese Übergabe innerhalb von `boost::fusion::filter_view` geschieht, wird mit `boost::mpl::arg<1>` ein Platzhalter aus Boost.MPL verwendet, um eine Lambda-Funktion zu erstellen. Die Aufgabe von `boost::mpl::arg<1>` ist ähnlich der von **`boost::phoenix::place_holders::arg1`** aus Boost.Phoenix.
Das auf diese Weise erstellte Prädikat führt dazu, dass die View **`v`** lediglich auf die int- und bool-Elemente des Tuples verweist. Führen Sie Beispiel 50.3 aus, wird Ihnen `10` und `true` ausgegeben – mehr nicht.

Beispiel 50.4 Iteratorzugriff auf einen Fusion-Container

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/iterator.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  auto it = begin(t);
  std::cout << *it << '\n';
  auto it2 = advance<boost::mpl::int_<2>>(it);
  std::cout << std::boolalpha << *it2 << '\n';
}
```

Nachdem Sie mit `boost::fusion::tuple` einen Container und mit `boost::fusion::for_each()` einen Algorithmus kennengelernt haben, wird es Sie wenig überraschen, im Beispiel 50.4 auf Iteratoren zu treffen. So bietet Boost.Fusion Funktionen wie `boost::fusion::begin()` und `boost::fusion::advance()` an. Sie funktionieren ähnlich wie die entsprechenden Funktionen aus der Standardbibliothek.
Beachten Sie, dass die Anzahl der Schritte, um die der Iterator erhöht werden soll, als Template-Parameter an `boost::fusion::advance()` übergeben werden muss. Mit `boost::mpl::int_` wird wieder auf einen Typ aus Boost.MPL zugegriffen.
`boost::fusion::advance()` gibt einen Iterator mit einem anderen Typ zurück als der, der als Parameter übergeben wurde. Deswegen wird im Beispiel 50.4 ein zweiter Iterator **`it2`** erstellt. Es ist nicht möglich, den Rückgabewert von `boost::fusion::advance()` dem ersten Iterator **`it`** zuzuweisen.
Neben den im Beispiel vorgestellten Funktionen bietet Boost.Fusion weitere an wie `boost::fusion::end()`, `boost::fusion::distance()`, `boost::fusion::next()` oder `boost::fusion::prior()`, um mit Iteratoren zu arbeiten.
Wenn Sie Beispiel 50.4 ausführen, wird `10` und `true` ausgegeben.

Beispiel 50.5 Ein heterogener Vektor mit `boost::fusion::vector`

```
#include <boost/fusion/container.hpp>
```

```
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef vector<int, std::string, bool, double> vector_type;
  vector_type v{10, "Boost", true, 3.14};
  std::cout << at<boost::mpl::int_<0>>(v) << '\n';

  auto v2 = push_back(v, 'X');
  std::cout << size(v) << '\n';
  std::cout << size(v2) << '\n';
  std::cout << back(v2) << '\n';
}
```

Bisher haben Sie als einzigen heterogenen Container `boost::fusion::tuple` kennengelernt. Im Beispiel 50.5 wird mit `boost::fusion::vector` ein neuer Container vorgestellt.
`boost::fusion::vector` verhält sich wie ein Vektor. So können Sie über einen Index auf ein Element zugreifen. Das geschieht jedoch nicht über den Operator `operator[]`, sondern über die freistehende Funktion `boost::fusion::at()`. Der Index wird dabei als Template-Parameter übergeben. Da es sich dabei um einen Typ handeln muss, wird der Index wieder mit `boost::mpl::int_` gekapselt.
Dem Vektor soll außerdem ein Element vom Typ char hinzugefügt werden. Dazu wird auf die Funktion `boost::fusion::push_back()` zugegriffen. Ihr werden zwei Parameter übergeben: Der Vektor, dem ein Element hinzugefügt werden soll, und der entsprechende Wert.
Beachten Sie, dass `boost::fusion::push_back()` einen neuen Vektor zurückgibt. Der Vektor **v** wird nicht verändert. Stattdessen entsteht ein neuer Vektor, der eine Kopie des ursprünglichen Vektors ist, aber ein zusätzliches Element enthält.
Das Beispielprogramm gibt mit `boost::function::size()` die Größe der beiden Vektoren **v** und **v2** aus. Wenn Sie das Programm ausführen, wird Ihnen `4` und `5` ausgegeben. Außerdem wird mit `boost::fusion::back()` auf das letzte Element in **v2** zugegriffen, das ebenfalls ausgegeben wird – ein `X`.
Wenn Sie sich Beispiel 50.5 näher ansehen, stellen Sie fest, dass es zwischen `boost::fusion::tuple` und `boost::fusion::vector` keinen Unterschied gibt. `boost::fusion::tuple` ist gleichbedeutend mit `boost::fusion::vector`. Beispiel 50.5 funktioniert auch mit `boost::fusion::tuple`.
Neben `boost::fusion::vector` bietet Boost.Fusion mit `boost::fusion::deque`, `boost::fusion::list` und `boost::fusion::set` weitere heterogene Container an. Im folgenden Beispiel wird Ihnen abschließend mit `boost::fusion::map` ein Container für Schlüssel/Wert-Paare vorgestellt.

Beispiel 50.6 Eine heterogene Map mit `boost::fusion::map`

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/fusion/algorithm.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  auto m = make_map<int, std::string, bool, double>("Boost", 10, 3.14, true);
  if (has_key<std::string>(m))
    std::cout << at_key<std::string>(m) << '\n';
  auto m2 = erase_key<std::string>(m);
  auto m3 = push_back(m2, make_pair<float>('X'));
  std::cout << std::boolalpha << has_key<std::string>(m3) << '\n';
}
```

Im Beispiel 50.6 wird eine heterogene Map mit Hilfe der Funktion `boost::fusion::map()` erstellt. Die Map hat den Typ `boost::fusion::map`, der dank des `auto`-Schlüsselworts nicht explizit angegeben werden muss.

In einer Map vom Typ `boost::fusion::map` sind wie bei einer `std::map` Schlüssel/Wert-Paare gespeichert. Die Schlüssel in der Fusion-Map sind jedoch Typen. Das heißt, ein Schlüssel/Wert-Paar besteht aus einem Typ und einem zugeordneten Wert. Dabei darf der Wert auf einem anderen Typ basieren als der Schlüssel. So ist im Beispiel 50.6 dem Schlüssel int der String „Boost“ zugewiesen.
Nachdem die Map erstellt wurde, wird mit `boost::fusion::has_key()` überprüft, ob ein Schlüssel `std::string` existiert. Im nächsten Schritt wird mit `boost::fusion::at_key()` auf das entsprechende Schlüssel/Wert-Paar zugegriffen. Da dem Schlüssel `std::string` die Zahl 10 zugewiesen ist, wird diese auf die Standardausgabe ausgegeben.
Das Schlüssel/Wert-Paar wird daraufhin mit `boost::fusion::erase_key()` gelöscht. Beachten Sie, dass sich die Map `m` nicht ändert. `boost::fusion::erase_key()` gibt eine neue Map zurück, in der das entsprechende Schlüssel/Wert-Paar fehlt.
Mit `boost::fusion::push_back()` wird der Map ein neues Schlüssel/Wert-Paar hinzugefügt. Der Schlüssel ist float, der Wert ein „X“. Für das Schlüssel/Wert-Paar wird auf die Funktion `boost::fusion::make_pair()` zugegriffen, die vergleichbar ist mit `std::make_pair()`.
Abschließend wird mit `boost::fusion::has_key()` überprüft, ob die Map einen Schlüssel `std::string` besitzt. Da dieser gelöscht wurde, wird `false` zurückgegeben.
Beachten Sie, dass Sie nicht mit `boost::fusion::has_key()` überprüfen müssen, ob ein Schlüssel existiert, bevor Sie `boost::fusion::at_key()` verwenden. Wenn Sie `boost::fusion::at_key()` einen Schlüssel übergeben, der in der entsprechenden Map nicht existiert, erhalten Sie einen Compilerfehler.

Beispiel 50.7 Fusion-Adapter für Strukturen

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <iostream>

struct strct
{
  int i;
  double d;
};

BOOST_FUSION_ADAPT_STRUCT(strct,
  (int, i)
  (double, d)
)

using namespace boost::fusion;

int main()
{
  strct s = {10, 3.14};
  std::cout << at<boost::mpl::int_<0>>(s) << '\n';
  std::cout << back(s) << '\n';
}
```

Boost.Fusion bietet mehrere Makros an, um Strukturen als Fusion-Container verwenden zu können. Dies ist möglich, da eine Struktur als heterogener Container angesehen werden kann. Beispiel 50.7 definiert eine Struktur, die mit Hilfe des Makros `BOOST_FUSION_ADAPT_STRUCT` zu einem Fusion-Container wird. So ist es möglich, mit Funktionen wie `boost::fusion::at()` oder `boost::fusion::back()` auf Elemente in der Struktur zuzugreifen.

Beispiel 50.8 Fusion-Unterstützung für `std::pair`

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <utility>
#include <iostream>

using namespace boost::fusion;

int main()
{
```

```
  auto p = std::make_pair(10, 3.14);
  std::cout << at<boost::mpl::int_<0>>(p) << '\n';
  std::cout << back(p) << '\n';
}
```

Boost.Fusion unterstützt Strukturen wie `std::pair` oder `boost::tuple`, ohne Makros verwenden zu müssen. Es muss lediglich die Headerdatei `boost/fusion/adapted.hpp` eingebunden werden.

Teil XII

Spracherweiterungen

Die folgenden Bibliotheken erweitern gewissermaßen die Programmersprache C++.

- Boost.Coroutine macht es möglich, Coroutinen in C++ einzusetzen – etwas, was andere Programmiersprachen üblicherweise über ein Schlüsselwort `yield` unterstützen.
- Boost.Foreach erlaubt den Einsatz einer range-basierten `for`-Schleife, wie sie mit C++11 Teil der Programmiersprache wurde.
- Mit Boost.Parameter können Parameter an Funktionen als Name/Wert-Paare und somit in einer beliebigen Reihenfolge übergeben werden – so wie es zum Beispiel Python erlaubt.
- Boost.Conversion bietet zwei Cast-Operatoren an, die `dynamic_cast` ersetzen und einen Downcast von einem Crosscast unterscheidbar machen.

Kapitel 51

Boost.Coroutine

Mit Boost.Coroutine ist es möglich, *Coroutinen* in C++ zu verwenden. Dabei handelt es sich um ein Feature, das in anderen Programmiersprachen häufig mit `yield` in Verbindung gebracht wird. In diesen Programmiersprachen ist es mit `yield` möglich, ähnlich wie mit `return` Funktionen zu beenden. Wird `yield` verwendet, merkt sich eine Funktion, wo sie beendet wurde. Wird die Funktion ein zweites Mal aufgerufen, setzt sie genau dort fort.
C++ kennt kein Schlüsselwort `yield`. Mit Boost.Coroutine ist es jedoch auch in C++ möglich, Funktionen zu beenden und später dort fortzusetzen, wo sie beendet wurden. Mit Boost.Asio existiert auch eine Boost-Bibliothek, die auf Boost.Coroutine zugreift und von Coroutinen profitiert.

> **Anmerkung**
>
> Boost.Coroutine ist eine relativ neue Bibliothek, die sich häufig ändert und inzwischen in mehreren Versionen vorliegt. Die erste Version erschien mit Boost 1.53.0. Mit Boost 1.55.0 wurde eine zweite Version eingeführt, die die erste Version ersetzte. Diese zweite Version wurde in Boost 1.56.0 erweitert. In diesem Kapitel lernen Sie die zweite Version kennen, wie sie mit Boost 1.55.0 eingeführt wurde. Sie können die Beispiele in diesem Kapitel kompilieren, wenn Sie Boost 1.55.0 oder eine neuere Version verwenden.

Beispiel 51.1 Coroutinen in Aktion

```
#include <boost/coroutine/all.hpp>
#include <iostream>

using namespace boost::coroutines;

void cooperative(coroutine<void>::push_type &sink)
{
  std::cout << "Hello";
  sink();
  std::cout << "world";
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  std::cout << ", ";
  source();
  std::cout << "!\n";
}
```

Beispiel 51.1 enthält neben `main()` eine Funktion `cooperative()`. `cooperative()` soll von `main()` als Coroutine aufgerufen werden. `cooperative()` soll daraufhin die Code-Ausführung vorzeitig beenden und zu `main()` zurückkehren, um dann ein zweites Mal aufgerufen zu werden und an der Stelle fortzusetzen, wo die Funktion vorher beendet wurde.

Um `cooperative()` als Coroutine verwenden zu können, wird auf zwei Typen pull_type und push_type zugegriffen, die beide von `boost::coroutines::coroutine` angeboten werden. `boost::coroutines::coroutine` ist ein Template, das im Beispiel mit void instanziiert wird.
Um Coroutinen verwenden zu können, benötigen Sie pull_type und push_type. Einen dieser Typen verwenden Sie, um ein Objekt zu erstellen, dem Sie die Funktion, die Sie als Coroutine verwenden wollen, als Parameter übergeben. Den anderen Typ verwenden Sie als ersten Parameter in der Funktion, die als Coroutine verwendet werden soll.
Im Beispiel wird in `main()` ein Objekt **`source`** vom Typ pull_type erstellt. Dem Konstruktor wird `cooperative()` übergeben. push_type wird als einziger Parameter im Funktionskopf von `cooperative()` verwendet. Der Konstruktor von pull_type ruft die Funktion, die als Parameter übergeben wird, als Coroutine auf. Im Beispiel wird demnach `cooperative()` aufgerufen, wenn **`source`** erstellt wird. Würde **`source`** auf push_type basieren, würde kein automatischer Aufruf stattfinden.
`cooperative()` gibt `Hello` auf die Standardausgabe aus. Anschließend greift die Funktion auf den Parameter **`sink`** zu, als würde es sich um eine Funktion handeln. Das ist möglich, weil push_type den Operator `operator()` überlädt. Während **`source`** in `main()` die Coroutine `cooperative()` repräsentiert, stellt **`sink`** in `cooperative()` die Funktion `main()` dar. Der Aufruf von **`sink`** führt dazu, dass `cooperative()` beendet und `main()` dort fortgesetzt wird, von wo `cooperative()` soeben aufgerufen worden war. In `main()` wird daraufhin ein Komma auf die Standardausgabe ausgegeben.
`main()` greift anschließend so auf **`source`** zu, als würde es sich um eine Funktion handeln. Das ist möglich, weil auch pull_type den Operator `operator()` überlädt. Daraufhin wird `cooperative()` aufgerufen und an der Stelle fortgesetzt, an der die Coroutine vorher beendet wurde. So gibt `cooperative()` `world` auf die Standardausgabe aus. Da es keine weiteren Anweisungen in `cooperative()` gibt, endet die Coroutine. Die Code-Ausführung kehrt zurück zu `main()`, wo ein Ausrufezeichen auf die Standardausgabe ausgegeben wird.
Beispiel 51.1 gibt zusammengenommen `Hello, world!` aus.
Sie können sich Coroutines als kooperative Threads vorstellen. In gewisser Weise laufen die Funktionen `main()` und `cooperative()` gleichzeitig. Code wird abwechselnd in `main()` und `cooperative()` ausgeführt. Die Ausführung der Anweisungen in den Funktionen findet zwar nacheinander statt. Dank Coroutines muss aber nicht mehr eine Funktion beendet werden, bevor eine andere ausgeführt werden kann.

Beispiel 51.2 Einen Wert von einer Coroutine zurückgeben

```
#include <boost/coroutine/all.hpp>
#include <functional>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<int>::push_type &sink, int i)
{
  int j = i;
  sink(++j);
  sink(++j);
  std::cout << "end\n";
}

int main()
{
  using std::placeholders::_1;
  coroutine<int>::pull_type source{std::bind(cooperative, _1, 0)};
  std::cout << source.get() << '\n';
  source();
  std::cout << source.get() << '\n';
  source();
}
```

Beispiel 51.2 ähnelt dem vorherigen. Das Template `boost::coroutines::coroutine` ist diesmal jedoch mit dem Typ int instanziiert. Damit ist es möglich, einen int-Wert von der Coroutine an den Aufrufer zu übergeben. Die Richtung, in der der int-Wert übergeben wird, hängt davon ab, wo pull_type und push_type verwendet werden. Im Beispiel wird ein Objekt vom Typ pull_type in `main()` instanziiert. `cooperative()` hat Zugriff auf ein Objekt vom Typ push_type. Da ausschließlich push_type verwendet werden kann, um einen Wert zu senden, und ausschließlich pull_type einen Wert empfangen kann, ist die Richtung der Datenübergabe vorgegeben.
Wenn `cooperative()` auf **`sink`** zugreift, muss ein int-Wert als Parameter übergeben werden. Dies ist zwingend

notwendig, weil **sink** auf dem Typ push_type basiert, der von der Klasse boost::coroutines::coroutine angeboten wird, die mit dem Template-Parameter int instanziiert wurde. Der Wert, der **sink** übergeben wird, kann über **source** in main() erhalten werden. Dazu bietet pull_type die Methode get() an.
Anhand des obigen Beispiels sehen Sie auch, wie Sie eine Funktion als Coroutine verwenden können, die mehrere Parameter erwartet. Da cooperative() einen zusätzlichen Parameter vom Typ int hat, kann die Funktion nicht direkt an den Konstruktor von pull_type übergeben werden. Im Beispiel wird std::bind() verwendet, um die Funktion mit pull_type zu verknüpfen.
Wenn Sie das Beispiel ausführen, wird 1 und 2 gefolgt von end ausgegeben.

Beispiel 51.3 Zwei Werte an eine Coroutine übergeben

```
#include <boost/coroutine/all.hpp>
#include <tuple>
#include <string>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<std::tuple<int, std::string>>::pull_type &source)
{
  auto args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
  source();
  args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
}

int main()
{
  coroutine<std::tuple<int, std::string>>::push_type sink{cooperative};
  sink(std::make_tuple(0, "aaa"));
  sink(std::make_tuple(1, "bbb"));
  std::cout << "end\n";
}
```

Beispiel 51.3 verwendet push_type in main() und pull_type in cooperative(): Die Datenübergabe findet vom Aufrufer zur Coroutine statt.
Anhand dieses Beispiels sehen Sie auch, wie Sie mehrere Werte übergeben. Boost.Coroutine unterstützt keine Datenübergabe von mehreren Werten, so dass Sie zum Beispiel ein Tuple verwenden müssen. Sie müssen selbst mehrere Werte in ein Tuple oder in eine andere Datenstruktur packen.
Das Beispiel gibt 0 aaa, 1 bbb und end auf die Standardausgabe aus.

Beispiel 51.4 Coroutinen und Ausnahmen

```
#include <boost/coroutine/all.hpp>
#include <stdexcept>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<void>::push_type &sink)
{
  sink();
  throw std::runtime_error("error");
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  try
  {
    source();
  }
  catch (const std::runtime_error &e)
  {
    std::cerr << e.what() << '\n';
```

```
  }
}
```

Wenn eine Ausnahme in einer Coroutine auftritt, wird die Coroutine sofort beendet. Die Ausnahme wird zum Aufrufer der Coroutine transportiert, wo sie abgefangen werden kann. Ausnahmen verhalten sich nicht anders als bei herkömmlichen Funktionsaufrufen.
Wenn Sie Beispiel 51.4 ausführen, wird `error` ausgegeben.

Kapitel 52

Boost.Foreach

Boost.Foreach bietet ein Makro an, das die mit C++11 in den Standard aufgenommene range-basierte `for`-Schleife simuliert. Mit dem Makro `BOOST_FOREACH`, das in der Headerdatei `boost/foreach.hpp` definiert ist, kann über Elemente in einer Sequenz iteriert werden, ohne auf Iteratoren zugreifen zu müssen. Arbeiten Sie mit einer Entwicklungsumgebung, die C++11 unterstützt, können Sie Boost.Foreach ignorieren.

Beispiel 52.1 `BOOST_FOREACH` und `BOOST_REVERSE_FOREACH` in Aktion

```
#include <boost/foreach.hpp>
#include <array>
#include <iostream>

int main()
{
  std::array<int, 4> a{{0, 1, 2, 3}};

  BOOST_FOREACH(int &i, a)
    i *= i;

  BOOST_REVERSE_FOREACH(int i, a)
  {
    std::cout << i << '\n';
  }
}
```

Das Makro `BOOST_FOREACH` erwartet zwei Parameter: Der erste Parameter ist eine Variable oder eine Referenz, der zweite Parameter eine Sequenz. Der Typ des ersten Parameters muss identisch sein mit dem Typ der Elemente in der Sequenz.

Als Sequenz zählt alles, was einen Iterator anbietet – zum Beispiel alle aus der Standardbibliothek bekannten Container. Boost.Foreach greift zwar nicht direkt auf die bekannten Methoden `begin()` und `end()` zu, sondern nimmt einen Umweg über die Bibliothek Boost.Range. Da diese Bibliothek jedoch auf Iteratoren aufsetzt, gilt grundsätzlich, dass alles, was Iteratoren anbietet, mit `BOOST_FOREACH` kompatibel ist.

Im Beispiel 52.1 wird mit `BOOST_FOREACH` über ein Array vom Typ `std::array` iteriert. Da der erste an `BOOST_FOREACH` übergebene Parameter eine Referenz ist, können in der Schleife Elemente im Array nicht nur gelesen, sondern auch geändert werden. So multipliziert die erste Schleife jede Zahl im Array mit sich selbst.

Die zweite Schleife greift auf das Makro `BOOST_REVERSE_FOREACH` zu, das genauso funktioniert wie `BOOST_FOREACH`, jedoch in umgekehrter Reihenfolge über eine Sequenz läuft. In dieser Schleife werden die Zahlen 9, 4, 1 und 0 in genau dieser Reihenfolge auf die Standardausgabe ausgegeben.

Sie können wie von Schleifen aus C++ gewohnt die geschweiften Klammern weglassen, wenn der Anweisungsblock hinter dem Schleifenkopf nur aus einer Anweisung besteht.

Beachten Sie, dass Sie innerhalb der Schleifen keine Operationen ausführen dürfen, die einen Iterator ungültig machen könnten. So dürfen Sie zum Beispiel nicht aus einem Vektor Elemente löschen oder ihm neue Elemente hinzufügen, während Sie über den Vektor iterieren. Letztendlich basieren `BOOST_FOREACH` und `BOOST_REVERSE_FOREACH` auf Iteratoren, die während der gesamten Iteration gültig sein müssen.

Kapitel 53

Boost.Parameter

Boost.Parameter macht es möglich, Parameter als Name/Wert-Paare zu übergeben. Dabei werden nicht nur Funktionsparameter unterstützt, sondern auch Template-Parameter. Die Bibliothek bietet sich besonders dann an, wenn lange Parameterlisten verwendet werden und es schwierig ist, sich die Reihenfolge und Bedeutung der Parameter zu merken. Name/Wert-Paare erlauben es, Parameter in einer beliebigen Reihenfolge zu übergeben. Da zu jedem Wert der Name des Parameters angegeben wird, ist außerdem die Bedeutung jedes Parameters im Code offensichtlich.

Beispiel 53.1 Funktionsparameter als Name/Wert-Paare

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char))
    (c, (double))
    (d, (std::string))
    (e, *)
  )
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_c = 3.14, _a = 1, _d = "Boost", _b = 'B', _e = true);
}
```

Beispiel 53.1 definiert eine Funktion `complicated()`, die fünf Parameter erwartet. Die Parameter sollen in einer beliebigen Reihenfolge übergeben werden können. Boost.Parameter bietet hierfür ein Makro namens `BOOST_`

PARAMETER_FUNCTION an, um eine derartige Funktion zu definieren.
Bevor BOOST_PARAMETER_FUNCTION verwendet werden kann, müssen die Parameter definiert werden, die für die Name/Wert-Paare verwendet werden sollen. Hierfür wird auf das Makro BOOST_PARAMETER_NAME zugegriffen. Sie übergeben diesem Makro lediglich einen Parameternamen. Im Beispiel wird BOOST_PARAMETER_NAME fünfmal verwendet, um die Parameternamen **a**, **b**, **c**, **d** und **e** zu definieren.
Beachten Sie, dass die Parameternamen automatisch in einem Namensraum tag definiert werden. Das soll Überschneidungen mit gleichnamigen Definitionen in einem Programm verhindern.
Nachdem die Parameternamen definiert wurden, wird auf BOOST_PARAMETER_FUNCTION zugegriffen, um eine Funktion complicated() zu definieren. Als erster Parameter wird der Typ des Rückgabewerts angegeben. Im Beispiel ist dies void. Beachten Sie, dass der Typ geklammert werden muss – der erste Parameter lautet (void). Als zweiter Parameter wird der Name der zu definierenden Funktion angegeben. Der dritte Parameter ist der Namensraum, in dem die Parameternamen definiert sind. Im vierten Parameter wird auf die Parameternamen zugegriffen, um sie näher zu spezifizieren.
Im Beispiel 53.1 beginnt der vierte Parameter mit required. Aus Sicht von Boost.Parameter handelt es sich dabei um ein Schlüsselwort, das angibt, dass die folgenden Parameter bei einem Aufruf der Funktion angegeben werden müssen.
Hinter required können ein oder mehrere Paare angegeben werden, deren erster Wert ein Parametername und deren zweiter Wert ein Typ ist. Auch hier ist zu beachten, dass der Typ geklammert werden muss.
Für die Parameternamen **a**, **b**, **c** und **d** wird auf verschiedene Typen zugegriffen. So kann zum Beispiel **a** verwendet werden, um einen int-Wert an complicated() zu übergeben. Für **e** ist kein Typ, sondern ein Sternchen angegeben. Boost.Parameter interpretiert dieses Sternchen dahingehend, dass jeder beliebige Typ erlaubt ist. **e** entspricht einem Template-Parameter.
Nachdem die verschiedenen Parameter an BOOST_PARAMETER_FUNCTION übergeben wurden, folgt der Funktionsblock. Dazu wird wie gewohnt auf ein Paar geschweifter Klammern zugegriffen. Im Funktionsblock kann auf die Parameternamen zugegriffen werden. Sie können wie Variablen verwendet werden, die den Typ haben, der ihnen innerhalb von BOOST_PARAMETER_FUNCTION zugewiesen wurde. Im Beispiel 53.1 werden die fünf Parameter auf die Standardausgabe ausgegeben.
Nachdem complicated() definiert wurde, wird die Funktion in main() aufgerufen. Für den Aufruf wird auf die verschiedenen Parameternamen zugegriffen und ihnen ein Wert zugewiesen. Dabei werden Parameter in einer beliebigen Reihenfolge übergeben.
Beachten Sie, dass Parameternamen bei einem Aufruf mit einem Unterstrich beginnen. Boost.Parameter versucht auf diese Weise zu verhindern, dass es zu Namensüberschneidungen kommt, weil andere Variablen möglicherweise wie die Parameter heißen.

> **Anmerkung**
>
> Um Funktionsparameter in C++ als Name/Wert-Paare anzugeben, können Sie auch auf das Named Parameter Idiom zurückgreifen. In diesem Fall ist eine Bibliothek wie Boost.Parameter nicht nötig.

Beispiel 53.2 Optionale Funktionsparameter

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
```

```
    (b, (char)))
  (optional
    (c, (double), 3.14)
    (d, (std::string), "Boost")
    (e, *, true))
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_b = 'B', _a = 1);
}
```

Mit `BOOST_PARAMETER_FUNCTION` können auch Funktionen mit optionalen Parametern definiert werden. Optionale Parameter können, müssen aber nicht beim Aufruf angegeben werden.
Im Beispiel 53.2 sind die Parameter **c**, **d** und **e** optional. Diese Parameter sind im Makro `BOOST_PARAMETER_FUNCTION` nicht hinter `required`, sondern hinter `optional` angegeben. Für Boost.Parameter ist `optional` ähnlich wie `required` ein Schlüsselwort, das optionale Parameter kennzeichnet.
Optionale Parameter werden ähnlich wie erforderliche Parameter definiert: Sie geben einen Parameternamen gefolgt von einem Typ an. Der Typ wird wie bereits gewohnt geklammert. Optionale Parameter erfordern als dritte Angabe einen Standardwert. Optionale Parameter werden auf diesen Standardwert gesetzt, wenn sie bei einem Aufruf nicht angegeben sind.
Beim Aufruf von `complicated()` werden die Parameter **a** und **b** übergegeben. Dies sind die einzig erforderlichen Parameter. Da die Parameter **c**, **d** und **e** nicht angegeben sind, werden sie auf Standardwerte gesetzt.
Boost.Parameter bietet neben `BOOST_PARAMETER_FUNCTION` weitere Makros an. So können Sie zum Beispiel das Makro `BOOST_PARAMETER_MEMBER_FUNCTION` verwenden, wenn Sie eine Methode definieren möchten. Mit `BOOST_PARAMETER_CONST_MEMBER_FUNCTION` können Sie eine konstante Methode definieren.
Beachten Sie, dass Sie mit Boost.Parameter auch Funktionen definieren können, die versuchen, Werte Parametern automatisch zuzuweisen. In diesem Fall müssen beim Aufruf keine Name/Wert-Paare angegeben werden – es reicht, einfach nur Werte anzugeben. Wenn sich die Typen der Werte, auf die Parameter gesetzt werden können, eindeutig unterscheiden, kann Boost.Parameter erkennen, welcher Wert welchem Parameter zugewiesen werden soll. Dies kann jedoch tiefergehende Kenntnisse in der Template Meta-Programmierung erfordern.

Beispiel 53.3 Template-Parameter als Name/Wert-Paare

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  required<tag::floating_point_type, std::is_floating_point<_>>,
  required<tag::any_type, std::is_object<_>>
> complicated_signature;

template <class A, class B, class C>
class complicated
```

```
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<int>,
    any_type<bool>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Im Beispiel 53.3 wird Boost.Parameter eingesetzt, um Template-Parameter als Name/Wert-Paare angeben zu können. Ähnlich wie bei Funktionen ist es so möglich, Template-Parameter in einer beliebigen Reihenfolge anzugeben.
Im Beispiel ist eine Klasse `complicated` definiert, die drei Template-Parameter erwartet. Da die Reihenfolge der Parameter keine Bedeutung hat, heißen sie `A`, `B` und `C`. `A`, `B` und `C` sind nicht die Namen der Parameter, die beim Zugriff auf die Klasse verwendet werden können. Diese werden ähnlich wie bei Funktionen mit einem Makro definiert. Für Template-Parameter heißt das Makro `BOOST_PARAMETER_TEMPLATE_KEYWORD`. So werden im Beispiel drei Parameternamen integral_type, floating_point_type und any_type definiert.
Nachdem die Parameternamen definiert sind, muss angegeben werden, welche Typen jeweils übergeben werden dürfen. So soll zum Beispiel der Parameter integral_type verwendet werden, um einen Typ wie `int` oder `long` zu übergeben, nicht aber einen Typ wie `std::string`. Dazu wird auf die Klasse `boost::parameter::parameters` zugegriffen. Diese wird verwendet, um eine Signatur zu erstellen. In der Signatur wird auf die definierten Parameternamen zugegriffen und angeben, welche Typen mit ihnen übergeben werden können.
`boost::parameter::parameters` ist ein Tuple, das Parameter beschreibt. Erforderliche Parameter werden dabei mit `boost::parameter::required` gekennzeichnet.
`boost::parameter::required` erfordert seinerseits zwei Parameter: Die erste Angabe ist der Name eines Parameters, der mit `BOOST_PARAMETER_TEMPLATE_KEYWORD` definiert wurde. Die zweite Angabe ist die Beschreibung des Typs, auf den der Parameter gesetzt werden darf. So ist zum Beispiel für den Parameternamen integral_type angegeben, dass er auf einen integralen Typ gesetzt werden muss. Diese Bedingung wird mit `std::is_integral<_>` ausgedrückt. Dabei handelt es sich um eine Lambda-Funktion, die auf Boost.MPL basiert. Diese Bibliothek stellt mit `boost::mpl::placeholders::_` einen Platzhalter zur Verfügung. Wenn der Typ, auf den der Parameter integral_type gesetzt ist, anstelle von `boost::mpl::placeholders::_` an `std::is_integral` übergeben wird und das Ergebnis dieser Lambda-Funktion wahr ist, handelt es sich um einen gültigen Typ. Die Bedingungen für die anderen Parameter floating_point_type und any_type sind ähnlich definiert.
Nachdem die Signatur erstellt und unter dem Name complicated_signature definiert wurde, wird innerhalb der Klasse `complicated` auf sie zugegriffen. Im ersten Schritt wird die Signatur mit `complicated_signature::bind` an die Template-Parameter `A`, `B` und `C` gebunden. Der neue Typ args stellt die Verbindung dar zwischen Template-Parametern, die beim Zugriff auf `complicated` angegeben werden, und den Bedingungen, die für diese Template-Parameter gelten. Im zweiten Schritt wird auf args zugegriffen, um die Parameter abzufragen. Dies erfolgt über `boost::parameter::value_type`. `boost::parameter::value_type` erhält als ersten Wert args, als zweiten einen Parameternamen. Der entsprechende Typ, der auf diese Weise definiert wird, repräsentiert den Typ, auf den ein Parameter gesetzt ist. So kann im Beispiel über integral_type auf den Typ zugegriffen werden, der mit dem Parameter integral_type als Template-Parameter `complicated` übergeben wird.
In `main()` wird auf `complicated` zugegriffen, um die Klasse zu instanziieren. Der Parameter integral_type wird auf int, floating_point_type auf double und any_type auf bool gesetzt. Die Reihenfolge, in der die Parameter angegeben werden, spielt keine Rolle. Anschließend wird auf die Typdefinitionen integral_type, floating_point_type und any_type zugegriffen, um ihre Werte mit `typeid` auszugeben. Wenn Sie das Beispiel mit Visual C++ 2013 kompilieren und ausführen, wird Ihnen `int`, `double` und `bool` angezeigt.

Beispiel 53.4 Optionale Template-Parameter

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
```

```
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  optional<tag::floating_point_type, std::is_floating_point<_>>,
  optional<tag::any_type, std::is_object<_>>
> complicated_signature;

template <class A, class B = void_, class C = void_>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type, float>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type, bool>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<short>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Beispiel 53.4 stellt basierend auf dem vorherigen optionale Template-Parameter vor. Dazu wird in der Signatur auf `boost::parameter::optional` zugegriffen. Außerdem werden die Template-Parameter von `complicated`, die optional sein sollen, auf `boost::parameter::void_` gesetzt. Für optionale Parameter wird `boost::parameter::value_type` außerdem ein Standardwert übergeben. Dies ist der Typ, auf den Parameter gesetzt werden, wenn sie nicht verwendet werden.
In `main()` wird wiederum auf `complicated` zugegriffen. Diesmal werden nur die beiden Parameter integral_type und floating_point_type verwendet. any_type ist optional. Wird das Beispiel mit Visual C++ 2013 kompiliert und ausgeführt, wird für integral_type `short`, für floating_point_type `double` und für any_type `bool` ausgegeben.
Ähnlich wie bei Funktionsparametern kann Boost.Parameter auch bei Template-Parametern eine automatische Erkennung vornehmen. Sie können eine Signatur so definieren, dass eine eindeutige Zuweisung von Typen an Parameter möglich ist. Ähnlich wie bei Funktionsparametern sind auch in diesem Fall tiefergehende Kenntnisse in der Template Meta-Programmierung erforderlich.

Kapitel 54

Boost.Conversion

Die Bibliothek Boost.Conversion bietet in der Headerdatei `boost/cast.hpp` die zwei Cast-Operatoren `boost::polymorphic_cast` und `boost::polymorphic_downcast` an. Sinn und Zweck dieser Cast-Operatoren ist es, eine dynamische Typumwandlung, die normalerweise mit `dynamic_cast` erfolgt, zu präzisieren.

Beispiel 54.1 Down- und Crosscasts mit `dynamic_cast`

```
struct base1 { virtual ~base1() = default; };
struct base2 { virtual ~base2() = default; };
struct derived : public base1, public base2 {};

void downcast(base1 *b1)
{
  derived *d = dynamic_cast<derived*>(b1);
}

void crosscast(base1 *b1)
{
  base2 *b2 = dynamic_cast<base2*>(b1);
}

int main()
{
  derived *d = new derived;
  downcast(d);

  base1 *b1 = new derived;
  crosscast(b1);
}
```

Im Beispiel 54.1 wird der Cast-Operator `dynamic_cast` zweimal eingesetzt: In der Funktion `downcast()` wandelt er den Zeiger auf eine Elternklasse in den Zeiger auf eine Kindklasse um. In der Funktion `crosscast()` wandelt er den Zeiger auf eine Elternklasse in einen Zeiger auf eine andere Elternklasse um. Die erste Umwandlung wird als Downcast bezeichnet, die zweite Umwandlung als Crosscast.
Indem die Cast-Operatoren von Boost.Conversion eingesetzt werden, kann wie im Beispiel 54.2 ein Downcast von einem Crosscast im Code unterschieden werden.

Beispiel 54.2 Down- und Crosscasts mit `polymorphic_downcast` und `polymorphic_cast`

```
#include <boost/cast.hpp>

struct base1 { virtual ~base1() = default; };
struct base2 { virtual ~base2() = default; };
struct derived : public base1, public base2 {};

void downcast(base1 *b1)
{
  derived *d = boost::polymorphic_downcast<derived*>(b1);
}

void crosscast(base1 *b1)
```

```
{
  base2 *b2 = boost::polymorphic_cast<base2*>(b1);
}

int main()
{
  derived *d = new derived;
  downcast(d);

  base1 *b1 = new derived;
  crosscast(b1);
}
```

Der Cast-Operator `boost::polymorphic_downcast` kann nur für Downcasts verwendet werden. Er verwendet intern `static_cast`, um die Umwandlung durchzuführen. Da `static_cast` nicht dynamisch überprüft, ob die Typumwandlung gültig ist, darf `boost::polymorphic_downcast` nur angewandt werden, wenn sicher ist, dass die Typumwandlung durchgeführt werden darf. Zur Kontrolle greift `boost::polymorphic_downcast` in Debug-Versionen auf `dynamic_cast` zu und überprüft mit `assert()`, ob die Typumwandlung erfolgen darf. Diese Überprüfung findet jedoch nur statt, wenn das Makro `NDEBUG` nicht definiert ist, was üblicherweise nur in Debug-Versionen der Fall ist.
Während mit `boost::polymorphic_downcast` ein Downcast möglich ist, muss `boost::polymorphic_cast` für einen Crosscast verwendet werden. Dieser Cast-Operator verwendet intern `dynamic_cast`, da dies der einzige Cast-Operator ist, der tatsächlich einen Crosscast durchführen kann. Der Vorteil von `boost::polymorphic_cast` gegenüber `dynamic_cast` ist, dass `boost::polymorphic_cast` im Fehlerfall eine Ausnahme vom Typ `std::bad_cast` wirft. `dynamic_cast` hingegen gibt einen Nullzeiger zurück, wenn die Typumwandlung fehlschlägt. Anstatt den Rückgabewert selbst auf einen Nullzeiger zu überprüfen, wird die Überprüfung von `boost::polymorphic_cast` abgenommen.
Beide Cast-Operatoren `boost::polymorphic_downcast` und `boost::polymorphic_cast` können nur verwendet werden, wenn Zeiger umgewandelt werden müssen. Ansonsten muss auf `dynamic_cast` zugegriffen werden. So kann `boost::polymorphic_downcast`, das auf `static_cast` basiert, Objekte vom Typ einer Elternklasse nicht in Objekte vom Typ einer Kindklasse umwandeln. Es ergibt außerdem keinen Sinn, `boost::polymorphic_cast` einzusetzen, wenn andere Typen als Zeiger umgewandelt werden sollen, weil `dynamic_cast` bei einer derartigen Umwandlung im Fehlerfall selbst eine Ausnahme vom Typ `std::bad_cast` wirft.

Teil XIII

Fehlerverarbeitung

Mit Boost.System und Boost.Exception lernen Sie zwei Bibliotheken zur Fehlerverarbeitung kennen.

- Boost.System stellt Klassen zur Verfügung, um Fehler zu beschreiben und identifizierbar zu machen.
- Boost.Exception erlaubt es, Ausnahmen, nachdem sie geworfen wurden, um zusätzliche Informationen zu ergänzen, die die Fehlerbehandlung vereinfachen sollen.

Kapitel 55

Boost.System

Boost.System ist eine Bibliothek, die im Wesentlichen vier Klassen zur Identifikation von Laufzeitfehlern zur Verfügung stellt. Alle vier Klassen sind mit C++11 in die Standardbibliothek aufgenommen worden, so dass Sie in einer Entwicklungsumgebung, die C++11 unterstützt, Boost.System nicht verwenden müssen. Viele Boost-Bibliotheken greifen jedoch auf Boost.System zu, so dass Sie unter Umständen über andere Bibliotheken mit Boost.System in Berührung kommen.
Die einfachste Klasse ist `boost::system::error_code`, die betriebssystemspezifische Fehler repräsentiert. Da Betriebssysteme typischerweise Fehler durchnummerieren, speichert `boost::system::error_code` in einer int-Variablen einen Fehlercode. Die Klasse wird im Beispiel 55.1 verwendet, um einem Aufrufer mitzuteilen, wenn eine Funktion fehlschlägt.

Beispiel 55.1 `boost::system::error_code` in Aktion

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  std::cout << ec.value() << '\n';
}
```

Beispiel 55.1 definiert eine Funktion `fail()`, deren einzige Aufgabe es ist, einen Fehler zurückzugeben. Damit der Aufrufer von `fail()` erfährt, wenn die Funktion fehlschlägt, übergibt er als Parameter ein Objekt vom Typ `boost::system::error_code`. Der Parameter wird als Referenz übergeben, damit der Aufrufer überprüfen kann, ob `fail()` einen Fehler zugewiesen hat. Viele Funktionen, die von Boost-Bibliotheken angeboten werden und `boost::system::error_code` verwenden, sehen ähnlich aus. So bietet zum Beispiel Boost.Asio eine Funktion `boost::asio::ip::host_name()` an, der ein Objekt vom Typ `boost::system::error_code` übergeben werden kann.
Boost.System definiert zahlreiche Fehlercodes im Namensraum `boost::system::errc`. Im Beispiel 55.1 wird auf den Fehlercode `boost::system::errc::not_supported` zugegriffen, um ihn dem Parameter **`ec`** zuzuweisen. Da **`boost::system::errc::not_supported`** eine Zahl ist, **`ec`** jedoch ein Objekt vom Typ `boost::system::error_code`, wird die Funktion `boost::system::errc::make_error_code()` aufgerufen. Sie erstellt ein Objekt vom Typ `boost::system::error_code` mit dem entsprechenden Fehlercode.
In `main()` wird für **`ec`** die Methode `value()` aufgerufen. Sie gibt die im Objekt gespeicherte Zahl zurück. 0 bedeutet standardmäßig, dass kein Fehler vorliegt. Jeder andere Wert weist auf einen Fehler hin. Welche Bedeutung der entsprechende Fehlercode hat, hängt vom Betriebssystem ab. Sie müssen in der Dokumentation Ihres Betriebssystems nachschlagen, welchen Fehler welcher Code beschreibt.
Neben `value()` bietet die Klasse `boost::system::error_code` eine Methode `category()` an. Diese führt

zur zweiten Klasse, die Boost.System definiert: `category()` gibt ein Objekt vom Typ `boost::system::error_category` zurück.
Fehlercodes sind wie bereits erfahren einfach nur Zahlen. Während ein Betriebssystemhersteller wie Microsoft darauf achten kann, dass Zahlen jeweils eindeutig einen Betriebssystemfehler repräsentieren, wird es für Anwendungsentwickler bedeutend schwieriger sicherzustellen, dass Fehlercodes quer über alle existierenden Programme hinweg einmalig sind. Es müsste eine Datenbank geben, in der alle Softwareentwickler die von ihnen benötigten Fehlercodes eintragen, so dass andere Softwareentwickler nicht die gleichen Fehlercodes für ganz andere Problemfälle verwenden. Da derartiges nicht praktikabel ist, gibt es Fehlerkategorien.
Fehler vom Typ `boost::system::error_code` gehören einer Fehlerkategorie an, die über die Methode `category()` erhalten werden kann. Wird ein Fehler mit `boost::system::errc::make_error_code()` erstellt, gehört er automatisch der generischen Kategorie an. Das ist die Kategorie, in die Fehler fallen, die nicht explizit einer anderen Kategorie zugewiesen wurden.

Beispiel 55.2 `boost::system::error_category` in Aktion

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  std::cout << ec.value() << '\n';
  std::cout << ec.category().name() << '\n';
}
```

Wenn Sie wie im Beispiel 55.2 `category()` aufrufen, erhalten Sie die zum Fehler gehörende Kategorie. Es handelt sich hierbei um ein Objekt vom Typ `boost::system::error_category`. Es bietet nur sehr wenige Methoden an. Sie können zum Beispiel `name()` aufrufen, um den Namen der Kategorie zu erhalten. So gibt Beispiel 55.2 `generic` auf die Standardausgabe aus.
Sie können auf die generische Kategorie auch über die freistehende Funktion `boost::system::generic_category()` zugreifen.
Boost.System bietet eine zweite Kategorie an: Betriebssystemfehler gehören der Kategorie System an – eine Referenz auf ein entsprechendes Objekt wird von der freistehenden Funktion `boost::system::system_category()` zurückgegeben. Geben Sie den Namen dieser Kategorie auf den Bildschirm aus, lesen Sie `system`.
Fehler sind über den Fehlercode und die Fehlerkategorie eindeutig identifizierbar. Weil Fehlercodes lediglich pro Kategorie eindeutig sein müssen, sollten Sie, wenn Sie für Ihr eigenes Programm Fehlercodes definieren wollen, eine neue Kategorie erstellen. So können Sie Fehlercodes beliebig vergeben, ohne dass es versehentlich zu Überschneidungen mit Fehlercodes anderer Entwickler kommt.

Beispiel 55.3 Eigene Fehlerkategorien erstellen

```
#include <boost/system/error_code.hpp>
#include <string>
#include <iostream>

class application_category :
  public boost::system::error_category
{
public:
  const char *name() const noexcept { return "my app"; }
  std::string message(int ev) const { return "error message"; }
};

application_category cat;

int main()
```

```
{
  boost::system::error_code ec{129, cat};
  std::cout << ec.value() << '\n';
  std::cout << ec.category().name() << '\n';
}
```

Um eine eigene Fehlerkategorie zu definieren, müssen Sie eine neue Klasse erstellen und diese von `boost::system::error_category` ableiten. Dies erfordert, verschiedene Methoden zu implementieren, die durch die Schnittstelle von `boost::system::error_category` vorgegeben sind. Sie müssen auf alle Fälle zwei Methoden `name()` und `message()` definieren, wenn Sie verhindern wollen, dass Ihre Klasse abstrakt wird, da diese Methoden als rein virtuelle Methoden in der Klasse `boost::system::error_category` definiert sind. Andere Methoden können, müssen aber nicht definiert werden, da `boost::system::error_category` Standarddefinitionen enthält.
Während `name()` den Namen der Fehlerkategorie zurückgibt, wird `message()` aufgerufen, um für einen Fehlercode eine Fehlerbeschreibung zu erhalten. Üblicherweise wird dabei im Gegensatz zu Beispiel 55.3 der Parameter **ev** ausgewertet und je nach Fehlercode eine andere Fehlerbeschreibung zurückgegeben.
Sie können ein Objekt vom Typ der neu definierten Fehlerkategorie verwenden, um mit ihm einen Fehler zu initialisieren. So wird im Beispiel 55.3 der Fehler **ec** mit der neuen Fehlerkategorie `application_category` erstellt. Der Fehlercode 129 ist demnach kein generischer Fehler, sondern besitzt eine Bedeutung, die vom Entwickler der Fehlerkategorie festgelegt wurde.

> **Anmerkung**
>
> Um Beispiel 55.3 mit Visual C++ 2013 zu kompilieren, entfernen Sie das Schlüsselwort `noexcept`. Diese Version des Microsoft-Compilers unterstützt `noexcept` noch nicht.

Die Klasse `boost::system::error_code` besitzt eine Methode `default_error_condition()`, die zur nächsten Klasse führt, die Boost.System anbietet. Diese Methode gibt ein Objekt vom Typ `boost::system::error_condition` zurück. Dabei handelt es sich um eine Klasse, deren Schnittstelle fast identisch mit der von `boost::system::error_code` ist. Der einzige Unterschied ist die soeben erwähnte Methode `default_error_condition()`, die ausschließlich von `boost::system::error_code` zur Verfügung gestellt wird.

Beispiel 55.4 `boost::system::error_condition` in Aktion

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  boost::system::error_condition ecnd = ec.default_error_condition();
  std::cout << ecnd.value() << '\n';
  std::cout << ecnd.category().name() << '\n';
}
```

`boost::system::error_condition` sieht der Klasse `boost::system::error_code` zum Verwechseln ähnlich und wird genauso verwendet. So können Sie wie im Beispiel 55.4 auch für `boost::system::error_condition` die Methoden `value()` und `category()` aufrufen.
Während die Klasse `boost::system::error_code` für plattformabhängige Fehlercodes verwendet wird, wird für plattformunabhängige Fehlercodes auf `boost::system::error_condition` zugegriffen. Indem Sie die Methode `default_error_condition()` aufrufen, wird ein plattformabhängiger Fehlercode in einen plattformunabhängigen vom Typ `boost::system::error_condition` umgewandelt.

Sie können `boost::system::error_condition` verwenden, wenn Sie Fehler identifizieren möchten, die plattformübergreifend sind. Das kann zum Beispiel ein fehlgeschlagener Zugriff auf eine nicht-existierende Datei sein. Während Betriebssysteme unterschiedliche Schnittstellen zum Zugriff auf Dateien anbieten können und die Fehlercodes dieser Schnittstellen vom jeweiligen Betriebssystem abhängen, ist der Zugriff auf eine nicht-existierende Datei ein Fehler, der betriebssystemübergreifend ist. Der Fehlercode, der von der Betriebssystemschnittstelle zurückgegeben wird, wird in `boost::system::error_code` gespeichert. Der Fehlercode, der den fehlgeschlagenen Zugriff auf die nicht-existierende Datei beschreibt, wird in `boost::system::error_condition` gespeichert.
Die letzte Klasse, die Boost.System anbietet und hier vorgestellt werden soll, ist `boost::system::system_error`. Sie ist von der Klasse `std::runtime_error` abgeleitet und ist ein Ausnahmetyp. Sie kann verwendet werden, wenn ein Fehler vom Typ `boost::system::error_code` in einer Ausnahme transportiert werden soll.

Beispiel 55.5 `boost::system::system_error` in Aktion

```
#include <boost/system/error_code.hpp>
#include <boost/system/system_error.hpp>
#include <iostream>

using namespace boost::system;

void fail()
{
  throw system_error{errc::make_error_code(errc::not_supported)};
}

int main()
{
  try
  {
    fail();
  }
  catch (system_error &e)
  {
    error_code ec = e.code();
    std::cerr << ec.value() << '\n';
    std::cerr << ec.category().name() << '\n';
  }
}
```

Die Funktion `fail()` wurde dahingehend geändert, dass im Fehlerfall eine Ausnahme vom Typ `boost::system::system_error` geworfen wird. Wie Sie am Beispiel 55.5 sehen, kann in dieser Ausnahme ein Fehler vom Typ `boost::system::error_code` transportiert werden. Die Ausnahme wird in `main()` abgefangen, um wie zuvor den Fehlercode und die Fehlerkategorie auf die Standardausgabe auszugeben. Die bereits erwähnte Funktion `boost::asio::ip::host_name()` gibt es in einer Variante, die genau so funktioniert.

Kapitel 56

Boost.Exception

Die Bibliothek Boost.Exception stellt den Ausnahmetyp `boost::exception` zur Verfügung, mit dem es möglich ist, Daten einer Ausnahme hinzuzufügen, nachdem diese geworfen wurde. Die Klasse ist in der Headerdatei `boost/exception/exception.hpp` definiert. Da Boost.Exception Klassen und Funktionen über mehrere Headerdateien verteilt, wird im Folgenden auf `boost/exception/all.hpp` zugegriffen, ohne dass Headerdateien einzeln eingebunden werden müssen.
Boost.Exception unterstützt den seit C++11 im Standard verankerten Mechanismus, mit dem Ausnahmen von einem Thread zu einem anderen transportiert werden können. So steht zum Beispiel mit `boost::exception_ptr` eine Klasse zur Verfügung, die `std::exception_ptr` entspricht. Boost.Exception ist jedoch kein vollwertiger Ersatz für die Headerdatei `exception` aus der Standardbibliothek. So fehlt Boost.Exception zum Beispiel die Unterstützung für verschachtelte Ausnahmen vom Typ `std::nested_exception`.

> Anmerkung
>
> Um die Beispiele in diesem Kapitel mit Visual C++ 2013 zu kompilieren, entfernen Sie das Schlüsselwort `noexcept`. Diese Version des Microsoft-Compilers unterstützt `noexcept` noch nicht.

Beispiel 56.1 `boost::exception` in Aktion

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public boost::exception, public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    throw allocation_failed{};
  return c;
}

char *write_lots_of_zeros()
{
```

```
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Im Beispiel 56.1 wird in `main()` eine Funktion `write_lots_of_zeros()` aufgerufen, die eine Funktion `allocate_memory()` aufruft. `allocate_memory()` reserviert dynamisch Speicher. Die Funktion übergibt `std::nothrow` an `new` und überprüft den Rückgabewert von `new` auf 0. Schlägt eine Speicherreservierung fehl, wird eine Ausnahme vom Typ `allocation_failed` geworfen. `allocation_failed` ersetzt die Ausnahme `std::bad_alloc`, die standardmäßig von `new` im Falle einer fehlgeschlagenen Speicherreservierung geworfen wird.
Die Funktion `write_lots_of_zeros()` weist `allocate_memory()` an, den größtmöglichen Speicherblock zu reservieren. Dazu wird auf `max()` der Klasse `std::numeric_limits` zugegriffen wird. Das Beispiel versucht absichtlich, einen so großen Speicherblock zu reservieren, dass die Reservierung fehlschlägt.
`allocation_failed` ist sowohl von `boost::exception` als auch von `std::exception` abgeleitet. Die Ableitung von `std::exception` ist für Boost.Exception nicht notwendig. So könnte `allocation_failed` auch von einer Klasse aus einer anderen Klassenhierarchie abgeleitet worden sein, um sie in ein existierendes Framework einzubetten. Im Beispiel 56.1 wird auf die im Standard definierte Klassenhierarchie für Ausnahmen zugegriffen. Für Boost.Exception genügt es, wenn `allocation_failed` lediglich von `boost::exception` abgeleitet wäre.
Wird eine Ausnahme vom Typ `allocation_failed` geworfen, kann die Funktion `allocate_memory()` als Fehlerquelle identifiziert werden, da sie die einzige Funktion ist, die eine Ausnahme dieses Typs wirft. Würde `allocate_memory()` in einem größeren Programm eingesetzt und von verschiedenen Funktionen aufgerufen werden, reicht allein der Typ `allocation_failed` nicht aus. So wäre es für die Fehlersuche hilfreich zu wissen, welche Funktion `allocate_memory()` aufgerufen hat und für die fehlgeschlagene Speicherreservierung verantwortlich ist.
Die besondere Schwierigkeit ist, dass die Ausnahme in der Funktion `allocate_memory()` auftritt, `allocate_memory()` aber nicht weiß, wer sie aus welchem Grund aufgerufen hat. `allocate_memory()` kann die Ausnahme nicht selbst mit zusätzlichen Informationen anreichern. Dies kann nur im übergeordneten Kontext geschehen. Mit Boost.Exception können zusätzliche Daten einer bereits geworfenen Ausnahme hinzugefügt werden. Dazu muss mit `typedef` ein auf `boost::error_info` basierender Typ definiert werden – und zwar für alle Daten, die neu in eine Ausnahme hineingepackt werden sollen.
`boost::error_info` ist ein Template und erwartet zwei Parameter: Der erste Parameter ist ein *Tag*, der den neuen auf `boost::error_info` basierenden Typ identifizierbar machen soll. Hier wird üblicherweise eine Struktur angegeben, die einen eindeutigen Namen bekommt. Der zweite Parameter bezieht sich auf den Typ der Daten, die in der Ausnahme gespeichert werden sollen. Im Beispiel 56.1 ist mit `errmsg_info` ein neuer Typ definiert, der über die Struktur `tag_errmsg` identifizierbar ist und einen String vom Typ `std::string` speichern kann.
Auf `errmsg_info` wird im `catch`-Block der Funktion `write_lots_of_zeros()` zugegriffen. Diese Funktion fängt Ausnahmen vom Typ `boost::exception` ab und fügt ihnen mit Hilfe des überladenen Operators

operator<< zusätzliche Daten hinzu. Es wird ein Objekt vom Typ errmsg_info erstellt, das mit dem String „writing lots of zeros failed“ initialisiert wird. Auf diese Weise wird diese Information in die Ausnahme gepackt, die anschließend mit throw erneut geworfen wird.
Die Ausnahme beschreibt dank ihres Typs nicht nur eine fehlgeschlagene Speicherreservierung, sondern enthält außerdem die Information, dass die Speicherreservierung fehlschlug, als das Programm in write_lots_of_zeros() viele Nullen schreiben wollte. Diese zusätzliche Information kann die Fehlersuche in einem größeren Program, in dem von vielen verschiedenen Funktionen auf allocate_memory() zugegriffen wird, wesentlich erleichtern.
Um alle in der Ausnahme vorhandenen Informationen abzurufen, kann wie im Beispiel 56.1 im catch-Block von main() die Funktion boost::diagnostic_information() aufgerufen werden. Diese Funktion ruft für die Ausnahme, die als Parameter übergeben wird, nicht nur what() auf, sondern greift auch auf alle zusätzlich hinzugefügten Daten zu. boost::diagnostic_information() gibt einen String vom Typ std::string zurück, der zum Beispiel auf die Standardfehlerausgabe ausgegeben werden kann.
Beispiel 56.1 mit Visual C++ 2013 kompiliert gibt folgende Meldung aus:

```
Throw location unknown (consider using BOOST_THROW_EXCEPTION)
Dynamic exception type: struct allocation_failed
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

Es wird der Typ der Ausnahme angegeben, die mit what() erhaltene Fehlermeldung und die zusätzliche Beschreibung mit dem Namen des Tags.
Wäre die Klasse allocation_failed nicht von std::exception abgeleitet worden, würde boost::diagnostic_information() nicht versuchen, die Methode what() aufzurufen. Innerhalb der Funktion boost::diagnostic_information() wird überprüft, ob ein Ausnahmetyp von std::exception abgeleitet ist und what() aufgerufen werden darf.
Der Name der Funktion, die die Ausnahme vom Typ allocation_failed geworfen hat, fehlt in obiger Meldung. Stattdessen heißt es, dass die „throw location“ unbekannt ist.
Boost.Exception stellt ein Makro zur Verfügung, mit dem eine Ausnahme derart geworfen werden kann, dass nicht nur der Name der entsprechenden Funktion in der Ausnahme gespeichert wird, sondern zusätzliche Informationen wie der Name der Datei und die Zeilennummer.

Beispiel 56.2 Mehr Informationen mit BOOST_THROW_EXCEPTION

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
```

```
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Wenn anstatt von `throw` das Makro `BOOST_THROW_EXCEPTION` verwendet wird, wird die Ausnahme um zusätzliche Informationen wie Funktionsname, Dateiname und Zeile ergänzt. Das funktioniert jedoch nur, wenn der Compiler entsprechende Makros unterstützt. Während Makros wie `__FILE__` und `__LINE__` seit C++98 standardisiert sind, gibt es mit `__func__` erst seit C++11 ein standardisiertes Makro, um den Namen der aktuellen Funktion zu erhalten. Da zahlreiche Compilerhersteller vor C++11 ein Makro für den Funktionsnamen definiert hatten, versucht `BOOST_THROW_EXCEPTION`, den jeweils verwendeten Compiler zu erkennen und ein entsprechendes Makro zu verwenden. So gibt Beispiel 56.2 mit Visual C++ 2013 kompiliert folgende Meldung aus:

```
main.cpp(20): Throw in function char *__cdecl allocate_memory(unsigned int)
Dynamic exception type: class boost::exception_detail::clone_impl<struct
  boost::exception_detail::error_info_injector<struct allocation_failed> >
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

Im Beispiel 56.2 ist die Klasse `allocation_failed` nicht mehr von `boost::exception` abgeleitet. Das Makro `BOOST_THROW_EXCEPTION` greift auf eine Funktion `boost::enable_error_info()` zu, die erkennt, ob ein Ausnahmetyp von `boost::exception` abgeleitet ist und – wenn dies nicht der Fall ist – erstellt einen neuen Ausnahmetyp, der sowohl vom angegebenen Ausnahmetyp als auch von `boost::exception` abgeleitet ist. Das ist der Grund, warum obige Meldung als Ausnahmetyp nicht nur `allocation_failed` ausgibt.

Beispiel 56.3 Gezielter Zugriff mit `boost::get_error_info()`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}
```

```
char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << *boost::get_error_info<errmsg_info>(e);
  }
}
```

Im Beispiel 56.3 wird nicht wie zuvor `boost::diagnostic_information()` verwendet, sondern mit `boost::get_error_info()` gezielt auf die Fehlermeldung vom Typ `errmsg_info` zugegriffen. Da `boost::get_error_info()` einen Smartpointer vom Typ `boost::shared_ptr` zurückgibt, muss über das Sternchen auf die Fehlermeldung zugegriffen werden. Nur für den Fall, dass der Parameter, der an `boost::get_error_info()` übergeben wird, nicht vom Typ `boost::exception` ist, ist der Smartpointer leer. Werfen Sie alle Ausnahmen mit dem Makro `BOOST_THROW_EXCEPTION`, ist sichergestellt, dass der Ausnahmetyp von `boost::exception` abgeleitet ist – eine Überprüfung des Smartpointers auf 0 ist dann nicht notwendig.

Teil XIV

Zahlenverarbeitung

Die folgenden Bibliotheken drehen sich ganz allgemein um Zahlen.

- Boost.Integer bietet integrale Typen an, um zum Beispiel angeben zu können, wie viele Bytes genau von einer Variable belegt werden sollen.
- Boost.Accumulators stellt Akkumulatoren zur Verfügung, denen Sie Zahlen übergeben, um zum Beispiel den Durchschnittswert oder die Standardabweichung zu berechnen.
- Boost.MinMax bietet eine Funktion an, um eine kleinste und größte Zahl mit einem einzigen Funktionsaufruf zu ermitteln.
- Boost.Random stellt Generatoren für Zufallszahlen zur Verfügung.
- Boost.NumericConversion bietet einen Cast-Operator an, der gegen einen unbeabsichtigten Überlauf schützt.

Kapitel 57

Boost.Integer

Die Bibliothek Boost.Integer bietet eine Headerdatei `boost/cstdint.hpp` an, die spezialisierte Typen für Ganzzahlen definiert. Diese Definitionen stammen aus dem Standard C99. Es handelt sich hierbei um die 1999 verabschiedete Version des Standards zur Programmiersprache C. Da die erste Version des C++-Standards 1998 veröffentlicht wurde, fehlten die in C99 definierten Typen für Ganzzahlen im C++-Standard.
C99 definiert die Typen, die Sie im Folgenden kennenlernen werden, in der Headerdatei `stdint.h`. Diese Headerdatei wurde in C++11 übernommen. In C++ heißt sie `cstdint`. Wenn Ihre Entwicklungsumgebung C++11 unterstützt, können Sie auf `cstdint` zugreifen und müssen `boost/cstdint.hpp` nicht verwenden.

Beispiel 57.1 Typen für Ganzzahlen mit Angabe von Bits

```
#include <boost/cstdint.hpp>
#include <iostream>

int main()
{
  boost::int8_t i8 = 1;
  std::cout << sizeof(i8) << '\n';

#ifndef BOOST_NO_INT64_T
  boost::uint64_t ui64 = 1;
  std::cout << sizeof(ui64) << '\n';
#endif

  boost::int_least8_t il8 = 1;
  std::cout << sizeof(il8) << '\n';

  boost::uint_least32_t uil32 = 1;
  std::cout << sizeof(uil32) << '\n';

  boost::int_fast8_t if8 = 1;
  std::cout << sizeof(if8) << '\n';

  boost::uint_fast16_t uif16 = 1;
  std::cout << sizeof(uif16) << '\n';
}
```

Die in `boost/cstdint.hpp` definierten Typen stehen im Namensraum `boost` zur Verfügung. Die Typen lassen sich in drei Kategorien einteilen:

- Typen wie boost::int8_t und boost::uint64_t tragen die exakte Speichergröße im Namen. Während boost::int8_t aus genau 8 Bits besteht, belegt boost::uint64_t genau 64 Bits.
- Typen wie boost::int_least8_t und boost::uint_least32_t bestehen aus mindestens so vielen Bits wie im Namen enthalten. So ist es möglich, dass die Speichergröße von boost::int_least8_t bei mehr als 8 Bits und von boost::uint_least32_t bei mehr als 32 Bits liegt.
- Typen wie boost::int_fast8_t und boost::uint_fast16_t legen ebenfalls eine Mindestgröße fest. Ihre tatsächliche Speichergröße ist so gesetzt, dass Variablen dieser Typen schnell verarbeitet werden können. So gibt

Beispiel 57.1 mit Visual C++ 2013 auf einem 64-Bit Windows 7-System erstellt für `sizeof(uif16) 4` aus.

Beachten Sie, dass 64-Bit-Typen nicht auf allen Plattformen zur Verfügung stehen. Sie können über das Makro `BOOST_NO_INT64_T` feststellen, ob 64-Bit-Typen fehlen.

Beispiel 57.2 Weitere spezialisierte Typen für Ganzzahlen

```
#include <boost/cstdint.hpp>
#include <iostream>

int main()
{
  boost::intmax_t imax = 1;
  std::cout << sizeof(imax) << '\n';

  std::cout << sizeof(UINT8_C(1)) << '\n';

#ifndef BOOST_NO_INT64_T
  std::cout << sizeof(INT64_C(1)) << '\n';
#endif
}
```

Boost.Integer definiert mit boost::intmax_t und boost::uintmax_t zwei Typen, mit denen die größtmöglichen Ganzzahlen gespeichert werden können. boost::intmax_t und boost::uintmax_t basieren auf den größtmöglichen primitiven Typen, die auf einer Plattform zur Verfügung stehen. Wird Beispiel 57.2 mit Visual C++ 2013 kompiliert auf einem 64-Bit Windows 7-System ausgeführt, gibt `sizeof(imax) 8` aus. Der größte Typ für Ganzzahlen belegt demnach 64 Bits.

Boost.Integer stellt außerdem Makros zur Verfügung, um Ganzzahlen als Literale mit bestimmten Typen zu verwenden. Wird in C++-Quellcode eine Ganzzahl niedergeschrieben, hat diese standardmäßig den Typ int und belegt mindestens 4 Bytes. Makros wie `UINT8_C` und `INT64_C` ermöglichen es, eine Mindestgröße für eine Ganzzahl als Literal anzugeben. Beispiel 57.2 gibt für `sizeof(UINT8_C(1))` mindestens `1` und für `sizeof(INT64_C(1))` mindestens `8` aus.

Boost.Integer bietet neben `boost/cstdint.hpp` weitere Headerdateien an. Diese stellen vorwiegend Klassen zur Verfügung, wie sie zur Template-Metaprogrammierung eingesetzt werden.

Kapitel 58

Boost.Accumulators

Boost.Accumulators bietet Klassen an, die Proben oder Messwerte verarbeiten können. So kann zum Beispiel der größte oder kleinste Wert gefunden oder die Summe aller Werte gebildet werden. Während dies auch mit Algorithmen aus der Standardbibliothek möglich ist, unterstützt Boost.Accumulators zahlreiche Berechnungen aus der Statistik. So kann zum Beispiel der Mittelwert oder die Standardabweichung berechnet werden.
Die Bibliothek heißt Boost.Accumulators, weil ein wesentliches Konzept der *Akkumulator* ist. Ein Akkumulator ist vergleichbar mit einem Container, der Ergebnisse neu berechnet, wenn Sie einen neuen Wert übergeben. Der Wert wird nicht zwangsläufig im Akkumulator gespeichert. Stattdessen berechnet der Akkumulator Zwischenergebnisse ständig neu, wenn Sie ihn mit Werten füttern.
Die Bibliothek Boost.Accumulators besteht aus drei Teilen:

- Das Framework definiert die grundsätzliche Struktur der Bibliothek. Es stellt die Klasse `boost::accumulators::accumulator_set` zur Verfügung, die immer zum Einsatz kommt, wenn Sie Boost.Accumulators verwenden. Während Sie diese und einige andere Klassen aus dem Framework kennen müssen, müssen Sie sich mit den Details des Frameworks nur dann vertraut machen, wenn Sie eigene Akkumulatoren entwickeln möchten.

  Um Zugriff auf `boost::accumulators::accumulator_set` und andere Klassen des Frameworks zu erhalten, binden Sie die Headerdatei `boost/accumulators/accumulators.hpp` ein.

- Boost.Accumulators stellt zahlreiche Akkumulatoren zur Verfügung, die Berechnungen ausführen. Sie können auf diese Akkumulatoren zugreifen und sie sofort verwenden.

  Über die Headerdatei `boost/accumulators/statistics.hpp` haben Sie Zugriff auf alle angebotenen Akkumulatoren.

- Boost.Accumulators stellt Operatoren zur Verfügung, um zum Beispiel komplexe Zahlen vom Typ `std::complex` mit einem int-Wert multiplizieren oder zwei Vektoren addieren zu können. So bietet die Headerdatei `boost/accumulators/numeric/functional.hpp` Zugriff auf Operatoren für `std::complex`, `std::valarray` und `std::vector`. Sie müssen diese Headerdatei nicht selbst einbinden, da sie von den Headerdateien der Akkumulatoren eingebunden wird. Sie müssen jedoch die Macros `BOOST_NUMERIC_FUNCTIONAL_STD_COMPLEX_SUPPORT`, `BOOST_NUMERIC_FUNCTIONAL_STD_VALARRAY_SUPPORT` und `BOOST_NUMERIC_FUNCTIONAL_STD_VECTOR_SUPPORT` definieren, wenn die Operatoren für die entsprechenden Klassen verfügbar sein sollen.

Alle von Boost.Accumulators angebotenen Klassen und Funktionen befinden sich in `boost::accumulators` oder darunter liegenden Namensräumen. So sind zum Beispiel alle Akkumulatoren im Namensraum `boost::accumulators::tag` definiert.

Beispiel 58.1 Zählen mit `boost::accumulators::tag::count`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<int, features<tag::count>> acc;
```

```
  acc(4);
  acc(-6);
  acc(9);
  std::cout << count(acc) << '\n';
}
```

Im Beispiel 58.1 kommt mit `boost::accumulators::tag::count` ein einfacher Akkumulator zum Einsatz: Er zählt die Werte, die dem Akkumulator übergeben werden. So gibt das Beispielprogramm `3` aus, wenn Sie es ausführen.
Wenn Sie einen Akkumulator verwenden möchten, greifen Sie auf die Klasse `boost::accumulators::accumulator_set` zu. Bei dieser Klasse handelt es sich um ein Template, dem Sie als ersten Parameter den Typ übergeben, den Sie für die Werte verwenden möchten, die verrechnet werden sollen. Im Beispiel 58.1 ist dies int. Als zweiten Parameter geben Sie die Akkumulatoren an, die Sie verwenden möchten. Beachten Sie, dass Sie mehrere Akkumulatoren verwenden können. Der Name der Klasse `boost::accumulators::accumulator_set` deutet es an, dass beliebig viele Akkumulatoren verwaltet werden können.
Genau genommen geben Sie keine Akkumulatoren an, sondern *Features*. Mit Features legen Sie fest, was berechnet werden soll. Sie bestimmen das Was, nicht das Wie. So kann es für Features unterschiedliche Implementationen geben. Bei diesen Implementationen handelt es sich um Akkumulatoren.
Im Beispiel 58.1 wird mit `boost::accumulators::tag::count` angegeben, dass ein Akkumulator verwendet werden soll, der Werte zählt. Existieren mehrere Akkumulatoren, die Werte zählen können, wählt Boost.Accumulators einen Standardakkumulator aus.
Beachten Sie, dass Sie Features nicht direkt als zweiten Parameter an `boost::accumulators::accumulator_set` übergeben, sondern über `boost::accumulators::features`.
Wenn Sie ein Objekt vom Typ `boost::accumulators::accumulator_set` erstellt haben, können Sie es wie eine Funktion verwenden und über den Operator `operator()` Werte übergeben. Diese werden sofort von dem oder den verwendeten Akkumulatoren verrechnet. Die Werte, die Sie übergeben, müssen zu dem Typ passen, den Sie als ersten Parameter an `boost::accumulators::accumulator_set` übergeben haben.
Für jedes Feature existiert ein gleichnamiger *Extractor*. Es handelt sich dabei um eine Funktion, die Sie aufrufen können, um das aktuelle Ergebnis eines Akkumulators zu erhalten. Im Beispiel 58.1 wird der Extractor `boost::accumulators::count()` aufgerufen und ihm als einziger Parameter **`acc`** übergeben. `boost::accumulators::count()` gibt daraufhin 3 zurück.

Beispiel 58.2 Durchschnitt und Varianz mit `mean` und `variance` errechnen

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>> acc;
  acc(8);
  acc(9);
  acc(10);
  acc(11);
  acc(12);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Im Beispiel 58.2 werden die beiden Features `boost::accumulators::tag::mean` und `boost::accumulators::tag::variance` eingesetzt, um Durchschnitt und Varianz für fünf Werte zu errechnen. Wenn Sie das Beispielprogramm ausführen, wird `10` und `2` ausgegeben.
Die Varianz von 2 kommt zustande, weil Boost.Accumulators für jeden der fünf Werte ein Gewicht von 0,2 annimmt. Der mit `boost::accumulators::tag::variance` ausgewählte Akkumulator gehört zu denjenigen, die mit Gewichten arbeiten. Wird kein Gewicht explizit angegeben, erhält jeder Wert automatisch das gleiche Gewicht.

Beispiel 58.3 Gewichtete Varianz errechnen

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
```

```
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>, int> acc;
  acc(8, weight = 1);
  acc(9, weight = 1);
  acc(10, weight = 4);
  acc(11, weight = 1);
  acc(12, weight = 1);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Im Beispiel 58.3 wird mit int ein dritter Template-Parameter an `boost::accumulators::accumulator_set` übergeben. Dieser Parameter beschreibt den Typ der Gewichte. So kann in diesem Beispiel jeder Wert mit einer Ganzzahl gewichtet werden.
Boost.Accumulators verwendet Boost.Parameter, um zusätzliche Angaben wie Gewichte als Name/Wert-Paare zu übergeben. Der Parameter für Gewichte heißt **`weight`**. Sie können den Parameter wie eine Variable behandeln und ihm einen Wert zuweisen. Das Name/Wert-Paar wird als zusätzlicher Parameter hinter jedem Wert an den Akkumulator übergeben.
Im Beispiel hat der Wert 10 ein Gewicht von 4, während alle anderen Werte ein Gewicht von 1 haben. Der Durchschnitt ist unverändert 10, da Gewichte für die Berechnung des Durchschnitts keine Rolle spielen. Die Varianz ist nun jedoch 1,25. Sie hat sich im Vergleich zum vorherigen Beispiel verringert, da der mittlere Wert stärker gewichtet ist als die anderen.
Boost.Accumulators bietet zahlreiche weitere Akkumulatoren an. Diese werden ähnlich wie die oben vorgestellten Akkumulatoren verwendet. Die Dokumentation der Bibliothek enthält eine Referenz, um einen Überblick über die verfügbaren Akkumulatoren zu erhalten.

Kapitel 59

Boost.MinMax

Wenn Sie den kleinsten und größten Wert suchen und bisher `std::min()` und `std::max()` verwendet haben, bietet Ihnen Boost.MinMax einen Algorithmus an, der den kleinsten und größten Wert mit einem einzigen Funktionsaufruf findet. Der von Boost.MinMax angebotene Algorithmus ist effizienter als wenn Sie `std::min()` und `std::max()` aufrufen.
Boost.MinMax ist in C++11 eingegangen. Sie finden die Algorithmen aus dieser Boost-Bibliothek in der Headerdatei `algorithm`, wenn Ihre Entwicklungsumgebung C++11 unterstützt.

Beispiel 59.1 `boost::minmax()` in Aktion

```
#include <boost/algorithm/minmax.hpp>
#include <boost/tuple/tuple.hpp>
#include <iostream>

int main()
{
  int i = 2;
  int j = 1;

  boost::tuples::tuple<const int&, const int&> t = boost::minmax(i, j);

  std::cout << t.get<0>() << '\n';
  std::cout << t.get<1>() << '\n';
}
```

Sie verwenden `boost::minmax()`, wenn Sie den kleinsten und den größten zweier Werte finden möchten. Während `std::min()` und `std::max()` nur einen Wert zurückgeben, gibt `boost::minmax()` entsprechend zwei Werte zurück. Dies geschieht über ein Tuple vom Typ `boost::tuple`. Die erste im Tuple gespeicherte Referenz verweist auf den kleineren, die zweite Referenz auf den größeren Wert. Beispiel 59.1 gibt entsprechend `1` und `2` in dieser Reihenfolge aus.
`boost::minmax()` ist in der Headerdatei `boost/algorithm/minmax.hpp` definiert.

Beispiel 59.2 `boost::minmax_element()` in Aktion

```
#include <boost/algorithm/minmax_element.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  typedef std::array<int, 4> array;
  array a{{2, 3, 0, 1}};

  std::pair<array::iterator, array::iterator> p =
    boost::minmax_element(a.begin(), a.end());

  std::cout << *p.first << '\n';
  std::cout << *p.second << '\n';
}
```

So wie die Standardbibliothek Algorithmen zur Verfügung stellt, um das kleinste und größte Element in einem Container zu finden, bietet Boost.MinMax `boost::minmax_element()` an, um das kleinste und größte Element in einem Container mit einem einzigen Funktionsaufruf zu finden.
Beachten Sie, dass `boost::minmax_element()` als Ergebnis ein `std::pair` mit zwei Iteratoren zurückgibt. Der Rückgabewert unterscheidet sich demnach von `boost::minmax()`. Der erste Iterator zeigt auf das kleinste und der zweite Iterator auf das größte Element im Container. Beispiel 59.2 gibt demnach `0` und `3` in dieser Reihenfolge aus.
`boost::minmax_element()` ist in der Headerdatei `boost/algorithm/minmax_element.hpp` definiert.
Die Algorithmen `boost::minmax()` und `boost::minmax_element()` können auch mit einem dritten Parameter aufgerufen werden, der angibt, wie Werte verglichen werden sollen. `boost::minmax()` und `boost::minmax_element()` können demnach genauso verwendet werden wie die Algorithmen aus der Standardbibliothek.

Kapitel 60

Boost.Random

Die Bibliothek Boost.Random bietet zahlreiche Zufallsgeneratoren an, die es Ihnen erlauben zu wählen, auf welche Weise Zufallszahlen erzeugt werden sollen. In C++ konnten über die Funktion `std::rand()` aus der Headerdatei `cstdlib` auch bisher Zufallszahlen erzeugt werden. In diesem Fall hängt es jedoch von der Implementation der Standardbibliothek ab, auf welche Weise Zufallsfallen generiert werden.
Sie können auf alle Zufallsgeneratoren und sonstigen Klassen und Funktionen der Bibliothek Boost.Random zugreifen, indem Sie die Headerdatei `boost/random.hpp` einbinden.
Beachten Sie, dass große Teile der Bibliothek mit C++11 in die Standardbibliothek aufgenommen wurden. Unterstützt Ihre Entwicklungsumgebung C++11, können Sie die folgenden Beispielprogramme zu Boost.Random umschreiben, indem Sie auf die Headerdatei `random` und den Namensraum `std` zugreifen.

Beispiel 60.1 Pseudo-Zufallszahlen mit `boost::random::mt19937`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  std::cout << gen() << '\n';
}
```

Im Beispiel 60.1 wird auf den Zufallsgenerator `boost::random::mt19937` zugegriffen. Indem der Operator `operator()` aufgerufen wird, wird eine Zufallszahl generiert, die auf die Standardausgabe ausgegeben wird.
Die von `boost::random::mt19937` generierten Zufallszahlen sind Ganzzahlen. Es hängt vom Zufallsgenerator ab, ob Ganz- oder Kommazahlen erzeugt werden. Alle Zufallsgeneratoren definieren einen Typ `result_type`, über den der Typ der Zufallszahlen erhalten werden kann. `boost::random::mt19937::result_type` ist auf boost::uint32_t gesetzt.
Alle Zufallsgeneratoren bieten außerdem zwei Methoden `min()` und `max()` an. Sie geben die kleinste und größte Zahl zurück, die von einem Zufallsgenerator erzeugt werden kann.
Fast alle von Boost.Random angebotenen Zufallsgeneratoren sind *Pseudo-Zufallsgeneratoren*. Pseudo-Zufallsgeneratoren erzeugen keine wirklich zufälligen Zahlen. Sie basieren auf Algorithmen, die scheinbar zufällige Zahlen erzeugen. `boost::random::mt19937` ist einer der angebotenen Pseudo-Zufallsgeneratoren.
Pseudo-Zufallsgeneratoren müssen typischerweise initialisiert werden. Werden sie mit gleichen Zahlen initialisiert, generieren sie gleiche Zufallszahlen. So wird im Beispiel 60.1 der Rückgabewert von `std::time()` an den Konstruktor von `boost::random::mt19937` übergeben. Auf diese Weise soll sichergestellt werden, dass ein wiederholter Aufruf des Programms zu unterschiedlichen Zeitpunkten nicht zur Ausgabe der immer gleichen Zufallszahl führt.
Pseudo-Zufallsgeneratoren sind für die meisten Anwendungsfälle gut genug. Auch `std::rand()` basiert auf einem Pseudo-Zufallsgenerator, der mit `std::srand()` initialisiert werden muss. Boost.Random bietet jedoch auch einen Zufallsgenerator an, der echte Zufallszahlen generieren kann – vorausgesetzt, ein Betriebssystem bietet eine Zufallsquelle an, die echte Zufallszahlen erzeugt.

Beispiel 60.2 Echte Zufallszahlen mit `boost::random::random_device`

```
#include <boost/random/random_device.hpp>
#include <iostream>

int main()
{
  boost::random::random_device gen;
  std::cout << gen() << '\n';
}
```

Boost.Random bietet mit `boost::random::random_device` einen *nicht-deterministischen Zufallsgenerator* an. Es handelt sich hierbei um einen Zufallsgenerator, der echte Zufallszahlen erzeugt. Es gibt keinen Algorithmus, der initialisiert werden muss und Zufallszahlen berechnet. Eine Vorhersage von Zufallszahlen ist demnach unmöglich. Nicht-deterministische Zufallsgeneratoren werden zum Beispiel in sicherheitsrelevanten Anwendungen verwendet.
`boost::random::random_device` greift zur Generierung von Zufallszahlen auf Betriebssystemfunktionen zu. Wird wie im Beispiel 60.2 der Standardkonstruktor verwendet, verwendet `boost::random::random_device` unter Windows den Kryptografiedienstanbieter MS_DEF_PROV und unter Linux den Gerätepfad `/dev/urandom` als Zufallsquelle.
Möchten Sie eine andere Zufallsquelle verwenden, greifen Sie auf den Konstruktor von `boost::random::random_device` zu, der einen Parameter vom Typ `std::string` erwartet. Es hängt vom Betriebssystem ab, wie dieser Parameter interpretiert wird. Unter Windows muss es der Name eines Kryptografiedienstanbieters sein, unter Linux ein Gerätepfad.
Beachten Sie, dass Sie die Headerdatei `boost/random/random_device.hpp` einbinden müssen, wenn Sie auf die Klasse `boost::random::random_device` zugreifen möchten. Diese Klasse ist nicht über `boost/random.hpp` verfügbar.

Beispiel 60.3 Die Zufallszahlen 0 oder 1 mit `bernoulli_distribution`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  boost::random::bernoulli_distribution<> dist;
  std::cout << dist(gen) << '\n';
}
```

Im Beispiel 60.3 wird auf den Pseudo-Zufallsgenerator `boost::random::mt19937` zugegriffen. Darüber hinaus kommt eine *Distribution* zum Einsatz. Als Distributionen bezeichnet Boost.Random Klassen, die die Bandbreite der Zufallszahlen eines Zufallsgenerators in eine andere Bandbreite übertragen. Während Zufallsgeneratoren wie `boost::random::mt19937` eine eingebaute Unter- und Obergrenze für Zufallszahlen haben, die über `min()` und `max()` ermittelt werden kann, werden häufig Zufallszahlen innerhalb einer bestimmten Bandbreite benötigt.
Im Beispiel 60.3 soll der Wurf einer Münze simuliert werden. Da eine Münze nur zwei Seiten hat, soll der Zufallsgenerator 0 oder 1 zurückgeben. Eine Distribution, die nur eines von zwei Ergebnissen zurückgibt, ist `boost::random::bernoulli_distribution`.
Distributionen werden wie Zufallsgeneratoren verwendet: Sie rufen den Operator `operator()` auf, um eine Zufallszahl zu erhalten. Distributionen müssen Sie jedoch als Parameter einen Zufallsgenerator übergeben. Im Beispiel 60.3 verwendet **`dist`** den Zufallsgenerator **gen**, um 0 oder 1 zurückzugeben.

Beispiel 60.4 Zufallszahlen zwischen 1 und 100 mit `uniform_int_distribution`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
```

```
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  boost::random::uniform_int_distribution<> dist{1, 100};
  std::cout << dist(gen) << '\n';
}
```

Boost.Random bietet zahlreiche Distributionen an. Im Beispiel 60.4 wird eine oft benötigte Distribution vorgestellt: Mit `boost::random::uniform_int_distribution` kann die Bandbreite der zu generierenden Zufallszahlen vorgegeben werden. So gibt **`dist`** im Beispiel 60.4 eine Zahl zwischen 1 und 100 zurück.
Beachten Sie, dass 1 und 100 ebenfalls gültige Zufallszahlen sind, die von **`dist`** zurückgegeben werden können. Distributionen werden mit einschließenden Unter- und Obergrenzen initialisiert.
Neben `boost::random::bernoulli_distribution` und `boost::random::uniform_int_distribution` können Sie auf zahlreiche weitere Distributionen in Boost.Random zugreifen. Dazu zählen zum Beispiel Distributionen wie `boost::random::normal_distribution` oder `boost::random::chi_squared_distribution`, wie sie in der Stochastik Verwendung finden.

Kapitel 61

Boost.NumericConversion

Die Bibliothek Boost.NumericConversion kann eingesetzt werden, wenn Zahlen eines nummerischen Typs in Zahlen eines anderen nummerischen Typs umgewandelt werden sollen. Derartiges kann in C++ auch implizit geschehen, wie im Beispiel 61.1 zu sehen.

Beispiel 61.1 Implizite Konvertierung von int zu short

```
#include <iostream>

int main()
{
  int i = 0x10000;
  short s = i;
  std::cout << s << '\n';
}
```

Das Beispielprogramm wird ohne Fehler kompiliert, weil die Umwandlung von int zu short in C++ automatisch erfolgt. Obwohl es keinen Compilerfehler gibt und das Programm ausgeführt werden kann, kann das Ergebnis der Umwandlung nicht vorhergesagt werden. Das Problem ist, dass die Zahl `0x10000` in der Variablen **`i`** zu groß ist, um in einer short-Variablen gespeichert werden zu können. Das Ergebnis ist abhängig von der Implementation. So gibt Beispiel 61.1 mit Visual C++ 2013 kompiliert `0` aus. Der Zahlenwert in **`s`** unterscheidet sich demnach deutlich von dem in **`i`**.
Um sicherzustellen, dass es bei einer Umwandlung von Zahlen zu keinen derartigen Fehlern kommt, kann der Cast-Operator `boost::numeric_cast` eingesetzt werden.

Beispiel 61.2 Überläufe entdecken mit `boost::numeric_cast`

```
#include <boost/numeric/conversion/cast.hpp>
#include <iostream>

int main()
{
  try
  {
    int i = 0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << '\n';
  }
  catch (boost::numeric::bad_numeric_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

`boost::numeric_cast` wird genauso angewandt wie die aus C++ bekannten Cast-Operatoren. Dazu muss lediglich die entsprechende Headerdatei eingebunden werden, die in diesem Fall `boost/numeric/conversion/cast.hpp` heißt.
`boost::numeric_cast` führt die gleiche Umwandlung von Zahlen unterschiedlicher nummerischer Typen aus, wie es in C++ auch ohne Cast-Operator implizit geschieht. Der Unterschied ist, dass `boost::numeric_cast`

überprüft, ob die Umwandlung einer Zahl so ausgeführt werden kann, dass das Ergebnis gleich dem Ausgangswert ist. Für Beispiel 61.2 bedeutet dies, dass keine Umwandlung durchgeführt wird. Stattdessen wird eine Ausnahme vom Typ `boost::numeric::bad_numeric_cast` geworfen, weil `0x10000` zu groß ist, um in einer short-Variable gespeichert werden zu können.
Genaugenommen wird keine Ausnahme vom Typ `boost::numeric::bad_numeric_cast` geworfen, sondern vom Typ `boost::numeric::positive_overflow`. Dieser Ausnahmetyp beschreibt einen sogenannten Überlauf – in diesem Fall für positive Zahlen. So gibt es auch eine Klasse `boost::numeric::negative_overflow`, die einen Überlauf für negative Zahlen beschreibt. Sehen Sie sich dazu Beispiel 61.3 an.

Beispiel 61.3 Überlauf für negative Zahlen

```
#include <boost/numeric/conversion/cast.hpp>
#include <iostream>

int main()
{
  try
  {
    int i = -0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << '\n';
  }
  catch (boost::numeric::negative_overflow &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Boost.NumericConversion definiert weitere Ausnahmetypen. Da alle von `boost::numeric::bad_numeric_cast` abgeleitet sind, können sie alle mit dieser Klasse abgefangen werden. Da `boost::numeric::bad_numeric_cast` seinerseits von `std::bad_cast` abgeleitet ist, könnte ein `catch`-Handler auch Ausnahmen dieses Typs abfangen.

Teil XV

Anwendungsbibliotheken

Anwendungsbibliotheken bezieht sich auf Bibliotheken, die üblicherweise ausschließlich in der Entwicklung von eigenständigen Anwendungen und nicht in der Entwicklung von Bibliotheken zum Einsatz kommen.

- Boost.Log ist die Logging-Bibliothek in Boost.
- Boost.ProgramOptions ist eine Bibliothek zum Definieren und Auswerten von Kommandozeilenparametern.
- Mit Boost.Serialization können Objekte serialisiert und zum Beispiel in Dateien gespeichert und später wieder von ihnen geladen werden.
- Boost.Uuid wird verwendet, um UUIDs zu verarbeiten, wie sie in bestimmten Umgebungen eingesetzt werden.

Kapitel 62

Boost.Log

Boost.Log ist die Logging-Bibliothek in Boost. Sie unterstützt zahlreiche Backends, um Daten in verschiedenen Formaten zu loggen. Backends werden über Frontends angesprochen, die Dienste bündeln und Logeinträge auf unterschiedliche Weise an Backends weiterreichen. So existiert zum Beispiel ein Frontend, das Logeinträge asynchron in einem Thread weitergibt. Frontends können Filter besitzen, um bestimmte Logeinträge zu ignorieren. Und sie definieren, wie Logeinträge als String formatiert werden. All diese Funktionen lassen sich beliebig erweitern, was Boost.Log zu einer umfangreichen und mächtigen Bibliothek macht.

Beispiel 62.1 Backend, Frontend, Core und Logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  sources::logger lg;

  BOOST_LOG(lg) << "note";
  sink->flush();
}
```

Beispiel 62.1 stellt die wesentlichen Bestandteile von Boost.Log vor. Wenn Sie mit dieser Bibliothek arbeiten, kommen Sie mit Backends, Frontends, dem Core und Loggern in Berührung:

- Backends bestimmen, wohin Daten geschrieben werden. Im obigen Beispiel wird das Backend `boost::log::sinks::text_ostream_backend` verwendet. Dieses Backend muss mit einem Stream vom Typ `std::ostream` initialisiert werden und leitet Logeinträge an diesen weiter.
- Frontends stellen das Verbindungsstück zwischen dem Core und einem Backend dar. Sie übernehmen verschiedene Aufgaben, die deswegen nicht in jedem Backend implementiert werden müssen. So können zum Beispiel Filter zu einem Frontend hinzugefügt werden. Das Frontend kann dann bestimmte Logeinträge ignorieren und leitet sie nicht ans Backend weiter.

  Im obigen Beispiel wird das Frontend `boost::log::sinks::asynchronous_sink` verwendet. Ein Frontend muss zwingend verwendet werden, auch wenn wie hier keine Filter zum Einsatz kommen. `boost:`

:log::sinks::asynchronous_sink basiert auf einem Thread, um Logeinträge asynchron an ein Backend weiterzureichen. Dadurch wird unter Umständen eine höhere Performance erreicht. Schreibvorgänge finden jedoch verzögert statt.

- Der Core ist der zentrale Baustein in Boost.Log, durch den alle Logeinträge geroutet werden. Er ist als Singleton implementiert. Über boost::log::core::get() kann jederzeit ein Zeiger auf den Core erhalten werden.

  Es ist notwendig, Frontends dem Core hinzuzufügen, damit Logeinträge an sie weitergereicht werden. Ob Logeinträge überhaupt weitergereicht werden, hängt vom Filter im Core ab. Filter können also nicht nur in Frontends, sondern auch im Core registriert werden. Während der Filter im Core global ist, sind Filter in Frontends lokal. Wird also ein Logeintrag vom Core gefiltert, wird er an keinen Frontend weitergereicht. Wird ein Logeintrag in einem Frontend gefiltert, kann er durchaus von anderen Frontends verarbeitet und an deren Backends weitergereicht werden.

- Der Logger ist der Bestandteil von Boost.Log, mit dem Sie die meiste Zeit in Berührung kommen. Während Sie Backend, Frontend und Core nur verwenden, um das Logging-Framework zu initialisieren, greifen Sie immer dann auf einen Logger zu, wenn Sie einen Logeintrag vornehmen möchten. Der Logger leitet den Logeintrag an den Core weiter.

  Im obigen Beispielprogramm hat der Logger den Typ boost::log::sources::logger. Es handelt sich hierbei um den einfachsten Logger. Wenn Sie einen Logeintrag vornehmen möchten, greifen Sie auf das Makro BOOST_LOG zu und übergeben den Logger als Parameter. Den Logeintrag können Sie so vornehmen, wie wenn Sie Daten auf einen Stream vom Typ std::ostream ausgeben.

Beachten Sie, wie Backend, Frontend, Core und Logger im Detail zusammenspielen. Das Frontend boost::log::sinks::asynchronous_sink ist ein Template, das das Backend boost::log::sinks::text_ostream_backend als Template-Parameter erhält. Daraufhin wird eine Instanz des Frontends erstellt. Dazu wird auf boost::shared_ptr zugegriffen. Das ist notwendig, da Frontends nur über diesen Smartpointer im Core registriert werden können: Der Aufruf von boost::log::core::add_sink() verlangt einen boost::shared_ptr.
Da das Backend ein Template-Parameter des Frontends ist, kann es nur konfiguriert werden, nachdem die Instanz des Frontends erstellt wurde. Wie dies im Detail geschieht, hängt vom Backend ab. Das Backend boost::log::sinks::text_ostream_backend bietet die Methode add_stream() an, um Streams hinzuzufügen. So kann dieses Backend sogar mit mehr als einem Stream konfiguriert werden. Andere Backends bieten andere Methoden zur Konfiguration an. Hier hilft nur ein Blick in die Dokumentation.
Um Zugriff auf das Backend zu erhalten, bieten alle Frontends die Methode locked_backend() an. Die Methode heißt locked_backend(), weil sie einen Zeiger zurückgibt, der einen synchronisierten Zugriff auf das Backend bietet, solange der Zeiger existiert. Sie können über locked_backend() von verschiedenen Threads aus auf das Backend zugreifen, ohne sich um eine Synchronisierung kümmern zu müssen.
Einen Logger wie boost::log::sources::logger können Sie über den Standardkonstruktor erstellen. Er greift über boost::log::core::get() automatisch auf den Core zu, um Logeinträge weiterzureichen.
Sie können auf Logger auch ohne Makro zugreifen. Logger sind gewöhnliche Objekte mit Methoden, die Sie aufrufen können. Makros wie BOOST_LOG machen es jedoch einfacher, Logeinträge zu schreiben. Ohne Makros könnte ein Logeintrag nicht in einer Codezeile geschrieben werden.
Beachten Sie, dass Beispiel 62.1 am Ende der Funkion main() boost::log::sinks::asynchronous_sink::flush() aufruft. Da das Frontend asynchron ist und Logeinträge in einem Thread schreibt, ist dieser Aufruf zwingend notwendig. Er sorgt dafür, dass der Thread alle gepufferten Logeinträge ans Backlog weiterreicht und diese geschrieben werden. Ohne den Aufruf von flush() könnte das Beispielprogramm beendet werden, bevor note ausgegeben wird.

Beispiel 62.2 boost::sources::severity_logger mit einem Filter

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

bool only_warnings(const attribute_value_set &set)
{
```

```
  return set["Severity"].extract<int>() > 0;
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(&only_warnings);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG(lg) << "note";
  BOOST_LOG_SEV(lg, 0) << "another note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Beispiel 62.2 baut auf dem vorherigen auf, ersetzt jedoch den Logger `boost::sources::logger` mit `boost::sources::severity_logger`. Dieser Logger fügt jedem Logeintrag ein Attribut hinzu, das das Loglevel beschreibt. Sie können das Makro `BOOST_LOG_SEV` verwenden, um das Loglevel explizit zu setzen.
Der Typ des Loglevels hängt vom Template-Parameter ab, den Sie an `boost::sources::severity_logger` übergeben. Im obigen Beispiel ist dies int. Daher werden an `BOOST_LOG_SEV` Zahlen wie 0 und 1 übergeben. Wird `BOOST_LOG` verwendet, ist das Loglevel automatisch auf 0 gesetzt.
Beispiel 62.2 ruft außerdem `set_filter()` auf, um einen Filter im Frontend zu registrieren. Es handelt sich hierbei um eine Funktion, die für jeden Logeintrag aufgerufen wird. Gibt die Funktion `true` zurück, wird der Logeintrag ans Backend weitergereicht. Andernfalls wird der Logeintrag ignoriert. Das Beispielprogramm definiert eine entsprechende Filterfunktion `only_warnings()` mit einem Rückgabewert vom Typ bool.
`only_warnings()` erwartet als Parameter ein `boost::log::attribute_value_set`. Dies ist der Typ, der Logeinträge repräsentiert, während sie im Logging-Framework herumgereicht werden. Boost.Log kennt mit `boost::log::record` noch einen anderen Typ für Logeinträge. `boost::log::record` speichert jedoch hauptsächlich ein `boost::log::attribute_value_set` und bietet eine entsprechende Methode `attribute_values()` an, um eine Referenz auf das `boost::log::attribute_value_set` zu erhalten. Filterfunktionen erhalten direkt ein `boost::log::attribute_value_set` und keinen `boost::log::record`.
`boost::log::attribute_value_set` heißt so, weil die Klasse Schlüssel/Wert-Paare speichert. Sie können sich die Klasse als `std::unordered_map` vorstellen.
Logeinträge bestehen aus Attributen. Attribute haben einen Namen und sind auf einen Wert gesetzt. Sie können Attribute selbst erstellen. Attribute können aber auch automatisch erstellt werden – zum Beispiel von Loggern. Dies ist der Grund, warum Boost.Log überhaupt unterschiedliche Logger anbietet. `boost::log::sources::severity_logger` fügt jedem Logeintrag ein Attribut namens Severity hinzu. Dieses Attribut speichert das Loglevel. So ist es möglich, im Filter das Loglevel daraufhin zu überprüfen, ob es größer als 0 ist.
`boost::log::attribute_value_set` bietet verschiedene Methoden an, um auf Attribute zuzugreifen. Die Methoden ähneln denen von `std::unordered_map`. So ist zum Beispiel der Operator `operator[]` überladen. Wird ihm ein Attributname übergeben, gibt er den Wert des Attributs zurück. Existiert das Attribut nicht, wird es automatisch erstellt.
Attributnamen haben den Typ `boost::log::attribute_name`. Diese Klasse bietet jedoch einen Konstruktor an, der einen String akzeptiert, so dass der String im Beispiel 62.2 direkt an `operator[]` übergeben werden kann.
Attributwerte haben den Typ `boost::log::attribute_value`. Diese Klasse stellt verschiedene Methoden zur Verfügung, um den Wert des Attributs in dem ihm eigenen Typ zu erhalten. Da das Loglevel ein int-Wert ist, wird beim Aufruf von `extract()` int als Template-Parameter übergeben.
Neben `extract()` bietet `boost::log::attribute_value` die Methoden `extract_or_default()` und `extract_or_throw()` an. `extract()` gibt einen mit einem Standardkonstruktor initialisierten Wert zurück, wenn die Typumwandlung nicht gelingt – bei int 0. `extract_or_default()` gibt im Fehlerfall einen Standardwert zurück, der als Parameter an die Methode übergeben wird. `extract_or_throw()` wirft im Fehlerfall eine

Ausnahme vom Typ `boost::log::runtime_error`.
Boost.Log bietet mit `boost::log::visit()` auch eine Visitor-Funktion an. Sie können diese Funktion anstelle von `extract()` verwenden, wenn Sie typsicher auf den Attributwert zugreifen möchten.
Wenn Sie Beispiel 62.2 ausführen, wird `warning` ausgegeben. Diese Meldung ist die einzige, die einen Loglevel größer als 0 hat und daher vom Filter nicht ausgesiebt wird.

Beispiel 62.3 Mit `set_formatter()` die Formatierung eines Logeintrags ändern

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

void severity_and_message(const record_view &view, formatting_ostream &os)
{
  os << view.attribute_values()["Severity"].extract<int>() << ": " <<
    view.attribute_values()["Message"].extract<std::string>();
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_formatter(&severity_and_message);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Beispiel 62.3 basiert auf dem vorherigen. Im Gegensatz zu diesem wird auch das Loglevel ausgegeben.
Frontends bieten eine Methode `set_formatter()` an, der eine Formatierungsfunktion übergeben werden kann. Wird ein Logeintrag von einem Frontend nicht gefiltert, wird er an die Formatierungsfunktion weitergereicht. Diese wandelt den Logeintrag in einen String um, der vom Frontend ans Backend weitergegeben wird. Rufen Sie `set_formatter()` nicht auf und definieren Sie keine Formatierungsfunktion, wird standardmäßig nur das ans Backend weitergereicht, was rechts von einem Makro wie `BOOST_LOG` steht.
Die Formatierungsfunktion, die im Beispiel 62.3 an `set_formatter()` übergeben wird, heißt `severity_and_message()`. Sie erwartet zwei Parameter vom Typ `boost::log::record_view` und `boost::log::formatting_ostream`. `boost::log::record_view` ist eine View auf einen Logeintrag. Eine View ist vergleichbar mit der bereits erwähnten Klasse `boost::log::record`. Der entscheidende Unterschied ist, dass eine View keine Änderungen am Logeintrag zulässt und daher konstant ist.
`boost::log::record_view` bietet eine Methode `attribute_values()` an, über die eine konstante Referenz auf das `boost::log::attribute_value_set` erhalten werden kann. `boost::log::formatting_ostream` ist der Stream, über den der String gebildet wird, der ans Backend weitergereicht werden soll.
Innerhalb von `severity_and_message()` wird auf die beiden Attribute Severity und Message zugegriffen. Die entsprechenden Werte werden mit `extract()` extrahiert und in den Stream geschrieben. Severity gibt wie bekannt das Loglevel als int zurück. Message ermöglicht einen Zugriff auf all das, was rechts von einem Makro wie `BOOST_LOG` steht. Dass die beiden Attribute Severity und Message heißen, muss der Dokumentation entnommen werden.
Beispiel 62.3 verwendet im Gegensatz zum vorherigen keinen Filter. Wenn Sie das Beispielprogramm ausführen, werden zwei Logeinträge geschrieben: `0:note` und `1:warning`.

Beispiel 62.4 Logeinträge filtern und formatieren mit Lambda-Funktionen

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::attr<int>("Severity") > 0);
  sink->set_formatter(expressions::stream <<
    expressions::attr<int>("Severity") << ": " << expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Beispiel 62.4 verwendet sowohl eine Filter- als auch eine Formatierungsfunktion. Die Funktionen sind diesmal als Lambda-Funktionen implementiert – nicht als C++11-Lambda-Funktionen, sondern als Lambda-Funktionen basierend auf Boost.Phoenix.

Boost.Log bietet im Namensraum `boost::log::expressions` Hilfsmittel für Lambda-Funktionen an. So können Sie über **`boost::log::expressions::stream`** auf den Stream und über **`boost::log::expressions::smessage`** auf das, was rechts von einem Makro wie `BOOST_LOG` steht, zugreifen. Sie können auch `boost::log::expressions::attr()` verwenden, um auf ein beliebiges Attribut zuzugreifen. So hätte im Beispiel anstelle von **`smessage`** auch `attr<std::string>("Message")` verwendet werden können.

Wenn Sie Beispiel 62.4 ausführen, wird `1:warning` und `2:error` ausgegeben.

Beispiel 62.5 Schlüsselwörter für Attribute definieren

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
```

```
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << severity << ": " <<
    expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Boost.Log ermöglicht es, eigene Schlüsselwörter zu definieren. So können Sie über das Makro `BOOST_LOG_ATTRIBUTE_KEYWORD` Schlüsselwörter erstellen, über die auf Attribute zugegriffen werden kann, ohne Attributnamen wiederholt als String an `boost::log::expressions::attr()` übergeben zu müssen.
Im Beispiel 62.5 wird auf das Makro `BOOST_LOG_ATTRIBUTE_KEYWORD` zugegriffen, um ein Schlüsselwort **`severity`** zu definieren. Das Makro erwartet drei Parameter: Zuerst den Namen des Schlüsselworts, dann den Namen des Attributs als String und abschließend den Typ des Attributs. In den Lambda-Funktionen, die im Beispiel für den Filter und die Formatierung eingesetzt werden, kann auf das neue Schlüsselwort zugegriffen werden. So ist es nicht nur möglich, auf von Boost.Log zur Verfügung gestellte Schlüsselwörter wie **`boost::log::expressions::smessage`** zuzugreifen – es können auch eigene definiert werden.
In allen bisherigen Beispielen waren Attribute automatisch vorhanden. Es konnte in Filter- oder Formatierungsfunktionen auf sie zugegriffen werden, weil Boost.Log sie automisch zur Verfügung stellte. Im Beispiel 62.6 sehen Sie, wie Sie eigene Attribute erstellen.

Beispiel 62.6 Eigene Attribute definieren

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << counter << " - " << severity <<
    ": " << expressions::smessage << " (" << timestamp << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});
```

```
  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

Sie erstellen globale Attribute, indem Sie die Methode `add_global_attribute()` für den Core aufrufen. Ein globales Attribut heißt so, weil es zu jedem Logeintrag automatisch hinzugefügt wird.
`add_global_attribute()` erwartet zwei Parameter: Sie müssen den Namen und den Typ des neuen Attributs angeben. Der Name wird als String übergeben. Für den Typ müssen Sie auf eine Klasse aus dem Namensraum `boost::log::attributes` zugreifen. So stellt Boost.Log zahlreiche Klassen zur Verfügung, die Attribute auf unterschiedliche Werte setzen können. Im Beispiel 62.6 wird die Klasse `boost::log::attributes::counter` verwendet, um ein Attribut namens LineCounter für jeden Logeintrag auf eine eindeutige Zahl zu setzen. Dieses neue Attribut nummeriert alle Logeinträge beginnend bei 1 durch.
Beachten Sie, dass `add_global_attribute()` keine Template-Methode ist. `boost::log::attributes::counter` wird nicht als Template-Parameter angegeben. Attributtypen müssen instanziiert und als Objekt übergeben werden.
Beispiel 62.6 verwendet ein zweites selbst erstelltes Attribut namens Timestamp. Dieses wird jedoch nicht als globales Attribut erstellt. Es handelt sich hierbei um ein Attribut mit einem begrenzten Gültigkeitsbereich.
Für das Attribut Timestamp kommt das Makro `BOOST_LOG_SCOPED_LOGGER_ATTR` zum Einsatz. Mit diesem Makro wird einem Logger ein Attribut hinzugefügt. Der Logger muss als erster Parameter ans Makro übergeben werden. Der zweite Parameter ist der Name des Attributs, der dritte Parameter der Typ. Als Typ ist für Timestamp `boost::log::attribute::local_clock` angegeben. Diese Klasse sorgt dafür, dass das Attribut Timestamp für jeden Logeintrag auf die aktuelle lokale Uhrzeit gesetzt wird.
Beachten Sie, dass das Attribut Timestamp ausschließlich dem Logeintrag „error“ hinzugefügt wird. Am Ende des Gültigkeitbereichs, in dem `BOOST_LOG_SCOPED_LOGGER_ATTR` verwendet wird, wird das Attribut automatisch vom Logger entfernt. `BOOST_LOG_SCOPED_LOGGER_ATTR` entspricht einem Aufruf von `add_attribute()` und `remove_attribute()`.
Wie im vorherigen Beispiel wird auch im Beispiel 62.6 auf das Makro `BOOST_LOG_ATTRIBUTE_KEYWORD` zugegriffen, um Schlüsselwörter für die neu erstellten Attribute zu definieren. In der Formatierungsfunktion wird auf die Schlüsselwörter zugegriffen, um Zeilennummer und Zeitpunkt auszugeben. Für Logeinträge, für die kein Attribut Timestamp definiert ist, ist **`timestamp`** leer und entspricht einem leeren String.

Beispiel 62.7 Hilfsfunktionen für Filter und Formatierungen

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <iomanip>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();
```

```
  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::is_in_range(severity, 1, 3));
  sink->set_formatter(expressions::stream << std::setw(5) << counter <<
    " - " << severity << ": " << expressions::smessage << " (" <<
    expressions::format_date_time(timestamp, "%H:%M:%S") << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

Boost.Log bietet zahlreiche Hilfsfunktionen für Filter und Formatierungen an. So wird im Beispiel 62.7 die Funktion `boost::log::expressions::is_in_range()` verwendet, um Logeinträge zu filtern, deren Loglevel nicht in eine bestimmte Bandbreite fällt. `boost::log::expressions::is_in_range()` erwartet das Attribut als ersten und die Unter- und Obergrenze als zweiten und dritten Parameter. Die Obergrenze ist der Wert, der gerade nicht mehr zur Bandbreite zählt.
In der Formatierungsfunktion kommt die Funktion `boost::log::expressions::format_date_time()` zum Einsatz. Sie kann verwendet werden, um einen Zeitpunkt zu formatieren. So wird diese Funktion im Beispiel 62.7 eingesetzt, um lediglich die Uhrzeit auszugegeben.
Beachten Sie, dass Sie in einer Formatierungsfunktion auch Stream-Manipulatoren aus der Standardbibliothek verwenden können. So kommt im Beispiel 62.7 `std::setw()` zum Einsatz, um eine Feldgröße für den LineCounter zu setzen.

Beispiel 62.8 Mehrere Logger, Frontends und Backends

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/channel_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/utility/string_literal.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <string>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(channel, "Channel", std::string)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend>
    ostream_sink;
  boost::shared_ptr<ostream_sink> ostream =
    boost::make_shared<ostream_sink>();
  boost::shared_ptr<std::ostream> clog{&std::clog,
    boost::empty_deleter{}};
```

```
  ostream->locked_backend()->add_stream(clog);
  core::get()->add_sink(ostream);

  typedef sinks::synchronous_sink<sinks::text_multifile_backend>
    multifile_sink;
  boost::shared_ptr<multifile_sink> multifile =
    boost::make_shared<multifile_sink>();
  multifile->locked_backend()->set_file_name_composer(
    sinks::file::as_file_name_composer(expressions::stream <<
    channel.or_default<std::string>("None") << "-" <<
    severity.or_default(0) << ".log"));
  core::get()->add_sink(multifile);

  sources::severity_logger<int> severity_lg;
  sources::channel_logger<> channel_lg{keywords::channel = "Main"};

  BOOST_LOG_SEV(severity_lg, 1) << "severity message";
  BOOST_LOG(channel_lg) << "channel message";
  ostream->flush();
}
```

Im Beispiel 62.8 werden mehrere Logger, Frontends und Backends gleichzeitig verwendet. So kommen neben den bekannten Klassen `boost::log::sinks::asynchronous_sink`, `boost::log::sinks::text_ostream_backend` und `boost::log::sources::severity_logger` mit `boost::log::sinks::synchronous_sink`, `boost::log::sinks::text_multifile_backend` und `boost::log::sources::channel_logger` ein neues Frontend, ein neues Backend und ein neuer Logger zum Einsatz.
Das Frontend `boost::log::sinks::synchronous_sink` bietet einen synchronisierten Zugriff auf ein Backend an. Somit ist es möglich, ein Backend in einer Multithreaded-Anwendung zu verwenden, selbst wenn das Backend nicht explizit für den Einsatz in einer Multithreaded-Anwendung entwickelt ist.
Der Unterschied zu `boost::log::sinks::asynchronous_sink` ist, dass `boost::log::sinks::synchronous_sink` nicht auf einem Thread basiert. Logeinträge werden im gleichen Thread ans Backend weitergereicht.
Im Beispiel 62.8 wird `boost::log::sinks::synchronous_sink` mit dem Backend `boost::log::sinks::text_multifile_backend` verwendet. Dieses Backend schreibt Logeinträge in eine oder mehrere Dateien. Dabei werden die Dateinamen nach einer Regel gebildet, die über `set_file_name_composer()` ans Backend übergeben wird. Wenn Sie wie im Beispiel die freistehende Funktion `boost::log::sinks::file::as_file_name_composer()` verwenden, können Sie die Regel mit den gleichen Bausteinen als Lambda-Funktion bilden, wie sie bei Formatierungsfunktionen zum Einsatz kommen. In diesem Fall wird aus Attributen kein String erstellt, der als Logeintrag geschrieben wird. Der String, der hier entsteht, ist der Dateiname, unter dem Logeinträge abgespeichert werden.
Im Beispiel 62.8 wird auf die beiden Schlüssewörter **`channel`** und **`severity`** zugegriffen. Diese sind mit Hilfe des Makros `BOOST_LOG_ATTRIBUTE_KEYWORD` definiert. Sie verweisen auf die gleichnamigen Attribute Channel und Severity. Der Zugriff erfolgt auf beide Schlüsselwörter über den Aufruf von `or_default()`. Diese Methode macht es möglich, einen Standardwert anzugeben, sollte ein Attribut nicht gesetzt sein. Wird ein Logeintrag geschrieben, ohne dass Channel und Severity gesetzt sind, wird er in einer Datei `None-0.log` gespeichert. Wird ein Logeintrag mit dem Loglevel 1 geschrieben, wird er in der Datei `None-1.log` gespeichert. Ein Logeintrag mit Loglevel 1 und Channel Main wird in der Datei `Main-1.log` gespeichert.
Das Channel-Attribut stammt vom Logger `boost::log::sources::channel_logger`. Dem Konstruktor dieses Channels muss ein Channel-Name übergeben werden. Der Name kann nicht direkt als String übergeben werden. Boost.Log erwartet, dass der Parameter über einen Namen eindeutig identifiziert wird. So ist es notwendig, `keywords::channel ="Main"` an den Konstruktor zu übergeben, auch wenn `boost::log::sources::channel_logger` keine anderen Parameter akzeptiert.
Beachten Sie, dass `boost::log::keywords::channel` nichts mit den Schlüsselwörtern zu tun hat, wie sie für Attribute verwendet werden.
Sinn und Zweck des Loggers `boost::log::sources::channel_logger` ist es, Logeinträge von verschiedenen Komponenten in einer Software klar zu trennen. So können Komponenten eigene Instanzen von `boost::log::sources::channel_logger` verwenden, die jeweils eindeutige Namen erhalten. Greifen Komponenten ausschließlich auf ihren Logger zu, lässt sich nachvollziehen, welche Logeinträge von welchen Komponenten in einer Anwendung stammen.

Beispiel 62.9 Ausnahmen im Logging-Framework zentral verarbeiten

```
#include <boost/log/common.hpp>
```

```
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/log/utility/exception_handler.hpp>
#include <boost/log/exceptions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

struct handler
{
  void operator()(const runtime_error &ex) const
  {
    std::cerr << "boost::log::runtime_error: " << ex.what() << '\n';
  }

  void operator()(const std::exception &ex) const
  {
    std::cerr << "std::exception: " << ex.what() << '\n';
  }
};

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);
  core::get()->set_exception_handler(
    make_exception_handler<runtime_error, std::exception>(handler{}));

  sources::logger lg;

  BOOST_LOG(lg) << "note";
}
```

Boost.Log bietet die Möglichkeit, Ausnahmen, die im Logging-Framework auftreten, zentral zu verarbeiten. So ist es nicht notwendig, jedes `BOOST_LOG` in einen `try`-Block zu setzen, um mit einem `catch` auf mögliche Ausnahmen zu reagieren.
Im Beispiel 62.9 wird die Methode `set_exception_handler()` aufgerufen. Der Core bietet diese Methode an, um einen Handler zu registrieren, an den Ausnahmen, die im Logging-Framework auftreten, weitergeleitet werden. Der Handler wird als Funktionsobjekt implementiert. Er muss einen oder mehrere überladene Operatoren `operator()` definieren, die Parameter mit entsprechenden Ausnahmetypen erwarten. Eine Instanz des Funktionsobjekts wird dann an `set_exception_handler()` übergeben – jedoch nicht direkt, sondern mit Hilfe der Template-Funktion `boost::log::make_exception_handler()`. Dieser Funktion müssen alle Ausnahmetypen, die der Handler verarbeiten soll, als Template-Parameter übergeben werden.
Boost.Log bietet mit `boost::log::make_exception_suppressor()` eine Funktion an, mit der alle Ausnahmen im Logging-Framework ignoriert werden können. Rufen Sie diese Funktion anstelle von `boost::log::make_exception_handler()` auf.

Beispiel 62.10 Makro zur Definition eines globalen Loggers

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
```

```
#include <exception>

using namespace boost::log;

BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(lg, sources::wlogger_mt)

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  BOOST_LOG(lg::get()) << L"note";
}
```

In allen bisherigen Beispielen wurden Logger lokal definiert. Möchten Sie einen Logger global definieren, greifen Sie wie im Beispiel 62.10 auf das Makro `BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT` zu. Übergeben Sie dem Makro als ersten Parameter den Namen und als zweiten Parameter den Typ des Loggers. Sie greifen auf den Logger nicht über den Namen, sondern über `get()` zu. Sie erhalten ähnlich wie beim Core einen Zeiger auf ein Singleton.

Boost.Log stellt neben `BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT` weitere Makros wie `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS` zur Verfügung, die weitergehende Initialisierungsmöglichkeiten anbieten. So können mit `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS` Parameter an den Konstruktor eines globalen Loggers übergeben werden. Allen Makros ist gemein, dass sie eine korrekte Initialisierung globaler Logger garantieren.

Boost.Log bietet weitere Funktionen an, für die sich ein Blick in die Dokumentation lohnt. So ist es zum Beispiel möglich, das Logging-Framework über einen Container zu konfigurieren, der Schlüssel/Wert-Paare als Strings speichert. In diesem Fall werden keine Objekte instanziiert und Methoden aufgerufen. Stattdessen kann zum Beispiel ein Schlüssel Destination auf Console gesetzt werden, was automatisch dazu führt, dass das Backend `boost::log::sinks::text_ostream_backend` verwendet wird. Über weitere Schlüssel kann das Backend konfiguriert werden. Da der Container auch als INI-Datei serialisiert werden kann, kann die Konfiguration in einer Textdatei erstellt und von dieser geladen werden.

Kapitel 63

Boost.ProgramOptions

Boost.ProgramOptions ist eine Bibliothek, die es einfach macht, Kommandozeilenparameter zu parsen. Der Einsatz von Boost.ProgramOptions ergibt daher nur dann Sinn, wenn Sie Kommandozeilenparameter parsen möchten. Dies ist üblicherweise in Konsolenanwendungen der Fall. Entwickeln Sie beispielsweise eine Windows-Anwendung mit grafischer Benutzeroberfläche, spielen Kommandozeilenparameter womöglich keine große Rolle.
Boost.ProgramOptions teilt die Aufgabe, Kommandozeilenparameter zu parsen, in drei Schritte ein:

1. Sie definieren Kommandozeilenparameter. Sie legen deren Namen fest und geben an, ob ein zusätzlicher Wert zum Namen auf der Kommandozeile angegeben werden muss. Falls der Kommandozeilenparameter als Name/Wert-Paar interpretiert wird, bestimmen Sie außerdem über einen Typ, welcher Art der Wert sein muss – also zum Beispiel ein String oder eine Zahl.

2. Sie verwenden einen Parser, dem Sie die Beschreibung Ihrer Kommandozeilenparameter und die Kommandozeile übergeben, mit der Ihre Anwendung gestartet wurde. Letztere erhalten Sie über zwei Parameter von `main()`, die üblicherweise **`argc`** und **`argv`** genannt werden.

3. Sie speichern die vom Parser ausgewerteten Kommandozeilenparameter ab. Boost.ProgramOptions bietet hierzu eine Klasse an, die von `std::map` abgeleitet ist. Kommandozeilenparameter werden also als Name/Wert-Paare gespeichert. Anschließend können Sie überprüfen, welche Parameter gespeichert wurden und auf welche Werte sie gesetzt sind.

Beispiel 63.1 zeigt, wie Sie grundsätzlich vorgehen müssen, um Kommandozeilenparameter mit Boost.ProgramOptions zu parsen.

Beispiel 63.1 Grundsätzliche Vorgehensweise bei Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <iostream>

using namespace boost::program_options;

void on_age(int age)
{
  std::cout << "On age: " << age << '\n';
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->default_value(3.14f), "Pi")
      ("age", value<int>()->notifier(on_age), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, desc), vm);
    notify(vm);
```

```
    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << vm["age"].as<int>() << '\n';
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Wenn Sie Boost.ProgramOptions verwenden möchten, reicht es, die Headerdatei `boost/program_options.hpp` einzubinden. Sie können dann auf alle Klassen und Funktionen dieser Bibliothek im Namensraum `boost::program_options` zugreifen.
Sie verwenden die Klasse `boost::program_options::options_description`, um Kommandozeilenparameter zu beschreiben. Da Sie eine Instanz dieser Klasse direkt auf einen Stream wie **`std::cout`** ausgeben können, um eine Beschreibung aller Kommandozeilenparameter anzuzeigen, können Sie dem Konstruktor einen Namen übergeben – also eine Art Überschrift für Ihre Kommandozeilenparameter.
Die Klasse `boost::program_options::options_description` besitzt eine Methode `add()`, die Sie aufrufen könnten, um die Beschreibung eines Kommandozeilenparameters in Form eines Objekts vom Typ `boost::program_options::option_description` zu übergeben. Möchten Sie viele Kommandozeilenparameter beschreiben, müssten Sie jedoch `add()` wiederholt aufrufen. Im Beispiel 63.1 wird stattdessen auf die Methode `add_options()` zugegriffen, die es einfacher macht, mehrere Kommandozeilenparameter zu definieren.
`add_options()` gibt ein Proxy-Objekt zurück, das die Instanz von `boost::program_options::options_description` repräsentiert. Der eigentliche Typ des Proxy-Objekts ist unwichtig. Interessanter ist, dass dieses Proxy-Objekt die Definition einer großen Zahl von Kommandozeilenparametern vereinfacht. So ist der Operator `operator()` derart überladen, dass Sie direkt die zur Definition eines Kommandozeilenparameters notwendigen Angaben übergeben können. Außerdem gibt dieser Operator eine Referenz auf das gleiche Proxy-Objekt zurück, so dass Sie mehrmals hintereinander auf `operator()` zugreifen können.
Im Beispiel 63.1 werden mit Hilfe des Proxy-Objekts drei Kommandozeilenparameter definiert. Der erste Kommandozeilenparameter ist `--help`. Der Beschreibungstext des Parameters lautet „Help screen“. Es handelt sich bei diesem Kommandozeilenparameter um einen Schalter, nicht um ein Name/Wert-Paar. `--help` ist auf der Kommandozeile entweder angegeben oder nicht. Hinter `--help` muss und kann keine zusätzliche Angabe vorgenommen werden.
Beachten Sie, dass der erste an `operator()` übergebene String „help,h“ lautet. Sie können für Kommandozeilenparameter eine Kurzversion angeben. Diese darf nur aus einem Buchstaben bestehen und wird hinter ein Komma gesetzt. Die Hilfe soll also nicht nur über `--help`, sondern auch über `-h` aufgerufen werden können.
Neben `--help` sind zwei weitere Kommandozeilenparameter definiert: `--pi` und `--age`. Diese unterscheiden sich von `--help` insofern, als dass es sich nicht um Schalter, sondern um Name/Wert-Paare handelt. Sowohl `--pi` als auch `--age` erwarten eine zusätzliche Angabe auf der Kommandozeile.
Um einen Kommandozeilenparameter als Name/Wert-Paar zu definieren, übergeben Sie als zweiten Parameter an `operator()` einen Zeiger auf ein Objekt vom Typ `boost::program_options::value_semantic`. Sie greifen dabei nicht direkt auf diese Klasse zu, sondern verwenden die Hilfsfunktion `boost::program_options::value()`. Diese erstellt ein Objekt vom Typ `boost::program_options::value_semantic` und gibt den Zeiger auf dieses zurück, den Sie direkt über `operator()` an das Proxy-Objekt weiterreichen.
Bei `boost::program_options::value()` handelt es sich um eine Template-Funktion. Sie übergeben den Typ des Werts, der dem Kommandozeilenparameter zugewiesen werden können soll, als Template-Parameter. Der Kommandozeilenparameter `--age` kann demnach auf eine Ganzzahl gesetzt werden, während für `--pi` eine Kommazahl akzeptiert wird.
Beachten Sie, dass das von `boost::program_options::value()` zurückgegebene Objekt einige nützliche Methoden anbietet. So können Sie zum Beispiel über `default_value()` eine Standardeinstellung vornehmen. Im Beispiel 63.1 ist `--pi` auf 3,14 gesetzt, wenn der Kommandozeilenparameter nicht explizit angegeben und die Standardeinstellung überschrieben wird.
Über `notifier()` kann eine Funktion mit dem Wert eines Parameters verknüpft werden. Dieser Funktion wird der entsprechende Wert übergeben, der von der Kommandozeile gelesen wurde. Auf diese Weise kann ein Wert zusätzlich verarbeitet werden. So ist im Beispiel 63.1 `on_age()` mit `--age` verknüpft. Wenn der Kommando-

zeilenparameter `--age` verwendet wird, um ein Alter anzugeben, wird dieses an die Funktion `on_age()` übergeben, von wo es auf die Standardausgabe ausgegeben wird.
Beachten Sie, dass die Verarbeitung eines Werts über eine Funktion wie `on_age()` optional ist. Sie müssen `notifier()` nicht verwenden, da Sie auch auf eine andere Weise an Parameterwerte gelangen.
Nachdem alle Kommandozeilenparameter definiert sind, verwenden Sie einen Parser. Boost.ProgramOptions stellt mit `boost::program_options::parse_command_line()` eine Hilfsfunktion zur Verfügung, der im Beispiel 63.1 mit **`argc`** und **`argv`** die Kommandozeile und mit **`desc`** die Definition der Kommandozeilenparameter übergeben wird. `boost::program_options::parse_command_line()` gibt die geparsten Kommandozeilenparameter in einem Objekt vom Typ `boost::program_options::parsed_options` zurück, auf das Sie gewöhnlich aber nicht zugreifen. Stattdessen übergeben Sie das Objekt direkt an `boost::program_options::store()`, um die geparsten Kommandozeilenparameter in einem Container abzulegen.
Im Beispiel 63.1 wird an `boost::program_options::store()` als zweiter Parameter **`vm`** übergeben. Es handelt sich dabei um ein Objekt vom Typ `boost::program_options::variables_map`. Diese Klasse ist von std::map<std::string, boost::program_options::variable_value> abgeleitet und bietet daher grundsätzlich die gleichen Methoden wie `std::map` an. So können Sie zum Beispiel mit `count()` überprüfen, ob ein bestimmter Kommandozeilenparameter verwendet wurde und im Container abgespeichert ist.
Bevor jedoch im Beispiel 63.1 auf **`vm`** zugegriffen und `count()` aufgerufen wird, wird die Funktion `boost::program_options::notify()` verwendet. Dies ist notwendig, damit Funktionen wie `on_age()`, die mit `notifier()` mit einem Parameter verknüpft wurden, aufgerufen werden. Ohne `boost::program_options::notify()` würde kein Aufruf von `on_age()` stattfinden.
Wenn Sie auf **`vm`** zugreifen, können Sie nicht nur mit `count()` überprüfen, ob ein Kommandozeilenparameter existiert. Sie können auch auf den Wert zugreifen, der dem Kommandozeilenparameter zugewiesen ist. Der Wert hat den Typ `boost::program_options::variable_value` – eine Klasse, die intern auf `boost::any` basiert. Sie können das Objekt vom Typ `boost::any` über eine Methode `value()` direkt erhalten.
Im Beispiel 63.1 wird nicht `value()` aufgerufen, sondern `as()`. Diese Methode gibt den entsprechenden Wert mit dem Typ zurück, den Sie an `as()` übergeben. `as()` greift auf `boost::any_cast()` zu, um die entsprechende Typumwandlung vorzunehmen.
Achten Sie darauf, dass der Typ, den Sie an `as()` übergeben, mit dem Typ des Parameterwerts übereinstimmt. Wenn wie im Beispiel 63.1 hinter dem Kommandozeilenparameter `--age` eine Ganzzahl vom Typ int angegeben werden muss, müssen Sie entsprechend int als Template-Parameter an `as()` übergeben.
Sie können Beispiel 63.1 auf unterschiedliche Weise ausführen. Rufen Sie das Programm zum Beispiel wie folgt auf:

```
test
```

In diesem Fall wird `Pi:3.14` ausgegeben. Auch wenn `--pi` nicht als Kommandozeilenparameter übergeben wird, ist dieser Kommandozeilenparameter immer vorhanden, nachdem im Programm ein Standardwert für Pi angegeben wurde.
Sie können den Parameterwert wie folgt überschreiben:

```
test --pi 3.1415
```

Die Ausgabe lautet nun `Pi:3.1415`.
Geben Sie zusätzlich ein Alter an:

```
test --pi 3.1415 --age 29
```

Sie erhalten `On age:29` und `Age:29`. Die erste Zeile wird ausgegeben, wenn im Programm `boost::program_options::notify()` aufgerufen wird, weil dann `on_age()` ausgeführt wird. Eine Ausgabe für `--pi` erhalten Sie nur deswegen nicht, weil im Beispiel `else if` verwendet wird und nur der erste gefundene Kommandozeilenparameter eine Aktion auslöst.
Rufen Sie nun die Hilfe auf:

```
test -h
```

Sie erhalten daraufhin eine vollständige Beschreibung aller Kommandozeilenparameter:

```
Options:
  -h [ --help ]         Help screen
  --pi arg (=3.1400001) Pi
  --age arg             Age
```

Sie können die Hilfe auf zweierlei Art und Weise aufrufen, da es eine Kurzversion des entsprechenden Kommandozeilenparameters gibt. Für `--pi` wird der Standardwert angezeigt.
Die Formatierung der Kommandozeilenparameter und ihrer Beschreibungen erfolgt automatisch. Sie müssen lediglich wie im Beispiel 63.1 das Objekt vom Typ `boost::program_options::options_description` auf die Standardausgabe ausgeben.
Rufen Sie das Beispiel nun wie folgt auf:

```
test --age
```

Sie erhalten folgende Ausgabe:

```
the required argument for option '--age' is missing.
```

Da hinter `--age` kein Wert angegeben ist, wirft der Parser, der in der Funktion `boost::program_options::parse_command_line()` zur Anwendung kommt, eine Ausnahme vom Typ `boost::program_options::error`. Diese Ausnahme wird im Beispiel abgefangen, um eine Meldung auf die Standardfehlerausgabe auszugeben.
`boost::program_options::error` ist von `std::logic_error` abgeleitet. Boost.ProgramOptions definiert weitere Ausnahmeklassen, die alle Kindklassen von `boost::program_options::error` sind. So könnte im Beispiel auch eine Ausnahme vom Typ `boost::program_options::invalid_syntax` abgefangen werden. Dies ist der exakte Typ der Ausnahme, die im Falle eines fehlenden Werts hinter dem Kommandozeilenparameter `--age` geworfen wird.
Nachdem Sie den grundsätzlichen Aufbau eines Programms kennengelernt haben, um mit Boost.ProgramOptions Kommandozeilenparameter zu parsen, lernen Sie im Beispiel 63.2 weitere Einstellungsmöglichkeiten kennen.

Beispiel 63.2 Spezielle Einstellungsmöglichkeiten mit Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(), std::ostream_iterator<std::string>{
    std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    int age;

    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->implicit_value(3.14f), "Pi")
      ("age", value<int>(&age), "Age")
      ("phone", value<std::vector<std::string>>()->multitoken()->
        zero_tokens()->composing(), "Phone")
      ("unreg", "Unrecognized options");

    command_line_parser parser{argc, argv};
    parser.options(desc).allow_unregistered().style(
      command_line_style::default_style |
      command_line_style::allow_slash_for_short);
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
```

```
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << age << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
    else if (vm.count("unreg"))
      to_cout(collect_unrecognized(parsed_options.options,
        exclude_positional));
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Im Beispiel 63.2 werden ähnliche Kommandozeilenparameter geparst wie zuvor. Es gibt jedoch einige Unterschiede. So wird zum Beispiel für den Parameterwert von Pi nicht mehr `default_value()`, sondern `implicit_value()` aufgerufen. Das bedeutet, dass Pi nicht mehr standardmäßig auf 3,14 gesetzt ist. Der Kommandozeilenparameter `--pi` muss gesetzt werden, damit Pi definiert ist. Wird `implicit_value()` verwendet, muss der Kommandozeilenparameter jedoch nicht auf einen Wert gesetzt werden. Es reicht, wenn `--pi` angegeben wird, ohne dass hinter dem Parameternamen ein Wert gesetzt wird. Dann ist Pi mit 3,14 definiert.
Der Funktion `boost::program_options::value()` wird für `--age` ein Zeiger auf eine Variable **`age`** übergeben. Auf diese Weise kann ein Parameterwert in einer Variablen gespeichert werden. Selbstverständlich steht der Parameterwert auch weiterhin im Container **`vm`** zur Verfügung.
Beachten Sie, dass der Parameterwert nur dann in der Variablen **`age`** gespeichert wird, wenn die Funktion `boost::program_options::notify()` aufgerufen wird. Diese Funktion wird nicht nur dann benötigt, wenn `notifier()` verwendet wird. Es ist grundsätzlich empfehlenswert, `boost::program_options::notify()` aufzurufen, nachdem mit `boost::program_options::store()` geparste Kommandozeilenparameter gespeichert wurden, da es andernfalls zu unvorhergesehenen Problemen kommen kann.
Beispiel 63.2 unterstützt mit `--phone` einen neuen Kommandozeilenparameter, um eine Telefonnummer an das Programm zu übergeben. Genauer gesagt sollen beliebig viele Telefonnummern an das Programm übergeben werden können. So soll das Programm zum Beispiel wie folgt aufgerufen werden können, um die beiden Telefonnummern 123 und 456 zu übergeben:

```
test --phone 123 456
```

Beispiel 63.2 unterstützt die Angabe mehrerer Telefonnummern auf diese Weise, weil für den Parameterwert `multitoken()` aufgerufen wurde. Da außerdem `zero_tokens()` aufgerufen wird, ist es auch möglich, `--phone` ohne Angabe einer Telefonnummer zu verwenden.
Das Beispielprogramm unterstützt auch folgende Möglichkeit zur Angabe mehrerer Telefonnummern:

```
test --phone 123 --phone 456
```

Auch in diesem Fall werden die beiden Telefonnummern 123 und 456 geparst. Der Grund ist, dass für den Parameterwert zusätzlich `composing()` aufgerufen wurde. Es ist dann erlaubt, einen Kommandozeilenparameter mehrfach zu verwenden – die Werte werden zusammengeführt.
Beachten Sie, dass der Typ für den Parameterwert für `--phone` std::vector<std::string> ist. Sie benötigen einen Container, damit mehrere Telefonnummern gespeichert werden können.
Mit `--unreg` ist im Beispiel 63.2 ein weiterer Kommandozeilenparameter definiert. Es handelt sich hierbei um einen Schalter, der entweder vorhanden ist oder nicht. Dieser Kommandozeilenparameter kann auf keinen Wert gesetzt werden. Er wird später im Beispielprogramm verwendet, um unbekannte und nicht in **`desc`** definierte Kommandozeilenparameter anzuzeigen.
Während im Beispiel 63.1 die Funktion `boost::program_options::parse_command_line()` verwendet wurde, um Kommandozeilenparameter zu parsen, wird im Beispiel 63.2 direkt auf einen Parser vom Typ `boost::program_options::command_line_parser` zugegriffen. Dem Konstruktor wird über **`argc`** und **`argv`** die Kommandozeile übergeben.

`boost::program_options::command_line_parser` bietet mehrere Methoden an. Sie müssen auf alle Fälle `options()` aufrufen, weil Sie mit dieser Methode dem Parser die Definitionen der Kommandozeilenparameter übergeben.
`options()` gibt wie auch alle anderen Methoden der Klasse eine Referenz auf den gleichen Parser zurück. So können mehrere Methodenaufrufe direkt nacheinander erfolgen. Im Beispiel 63.2 wird nach `options()` die Methode `allow_unregistered()` aufgerufen, um dem Parser mitzuteilen, dass er bei unbekannten Kommandozeilenparametern keine Ausnahme werfen soll. Über `style()` wird angegeben, dass für Kommandozeilenparameter in der Kurzversion auch der Schrägstrich verwendet werden darf. Sie können also die Hilfe nicht nur mit `-h`, sondern auch mit `/h` aufrufen.
Beachten Sie, dass `boost::program_options::parse_command_line()` einen vierten Parameter unterstützt, der an `style()` weitergereicht wird. Sie können diese Hilfsfunktion aufrufen und müssen nicht explizit auf den Parser zugreifen, wenn Sie eine Option wie **`boost::program_options::command_line_style::allow_slash_for_short`** verwenden möchten.
Nachdem Sie die gewünschten Einstellungen vorgenommen haben, rufen Sie für den Parser `run()` auf. Diese Methode gibt die geparsten Kommandozeilenparameter in einem Objekt vom Typ `boost::program_options::parsed_options` zurück, das Sie an `boost::program_options::store()` weiterreichen, um die Kommandozeilenparameter in **`vm`** zu speichern.
Im unteren Teil von Beispiel 63.2 wird wie zuvor auf **`vm`** zugegriffen, um Kommandozeilenparameter auszuwerten. Neu ist lediglich die Funktion `boost::program_options::collect_unrecognized()`, die für den Kommandozeilenparameter `--unreg` aufgerufen wird. Dieser Funktion muss das Objekt vom Typ `boost::program_options::parsed_options` übergeben werden, das von `run()` zurückgegeben wurde. Die Funktion gibt daraufhin alle unbekannten Kommandozeilenparameter in einem std::vector<std::string> zurück. Wenn Sie das Programm zum Beispiel mit **test --unreg --abc** aufrufen, wird `--abc` ausgegeben.
Beachten Sie, dass `boost::program_options::collect_unrecognized()` als zweiter Parameter **`boost::program_options::exclude_positional`** übergeben wird. Damit geben Sie an, dass Positionsparameter ignoriert werden sollen. Im Beispiel 63.2 spielt dies keine Rolle, weil keine Positionsparameter definiert sind. `boost::program_options::collect_unrecognized()` ist jedoch so definiert, dass Sie einen entsprechenden Parameter angeben müssen.
Was Positionsparameter sind, soll Ihnen anhand folgenden Beispiels gezeigt werden.

Beispiel 63.3 Positionsparameter bei Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(),
    std::ostream_iterator<std::string>{std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("phone", value<std::vector<std::string>>()->
        multitoken()->zero_tokens()->composing(), "Phone");

    positional_options_description pos_desc;
    pos_desc.add("phone", -1);

    command_line_parser parser{argc, argv};
    parser.options(desc).positional(pos_desc).allow_unregistered();
    parsed_options parsed_options = parser.run();
```

```
    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Beispiel 63.3 definiert `--phone` als Positionsparameter. Dazu wird auf die Klasse `boost::program_options::positional_options_description` zugegriffen. Diese Klasse bietet eine Methode `add()` an, der der Name eines Positionsparameters und eine Positionsnummer übergeben werden muss. Im Beispiel wird „phone“ und -1 übergeben.
Die Idee hinter Positionsparametern ist, dass Sie Werte auf der Kommandozeile angeben können, ohne sie hinter einen Kommandozeilenparameter stellen zu müssen. So können Sie Beispiel 63.3 wie folgt aufrufen:

```
test 123 456
```

Obwohl `--phone` nicht verwendet wird, werden 123 und 456 als Telefonnummern erkannt.
Wenn Sie die Methode `add()` für ein Objekt vom Typ `boost::program_options::positional_options_description` aufrufen, weisen Sie Angaben auf der Kommandozeile über die Positionsnummer einem Kommandozeilenparameter zu. Im obigen Fall hat 123 die Positionsnummer 0 und 456 die Positionsnummer 1. Da im Beispiel 63.3 -1 an `add()` übergeben wurde, werden alle Angaben `--phone` zugewiesen. Würde zum Beispiel 0 übergeben werden, würde ausschließlich 123 als Telefonnummer erkannt werden. Würde 1 übergeben werden, würde lediglich 456 als Telefonnummer geparst werden.
Beachten Sie, dass **`pos_desc`** mit `positional()` an den Parser übergeben wird. Nur dann weiß der Parser, welche Kommandozeilenparameter Positionsparameter sind.
Beachten Sie, dass Sie außerdem sicherstellen müssen, dass die von Ihnen verwendeten Positionsparameter definiert sind. Für Beispiel 63.3 bedeutet dies, dass „phone“ nur deswegen an `add()` übergeben werden kann, weil es eine Definition für den Kommandozeilenparameter `--phone` in **`desc`** gibt.
In allen Beispielen wurde Boost.ProgramOptions bisher verwendet, um Kommandozeilenparameter zu parsern. Die Bibliothek unterstützt jedoch auch das Laden von Parametern aus einer Datei. Speziell für den Fall, dass wiederholt die gleichen Kommandozeilenparameter angegeben werden müssen, kann eine Konfigurationsdatei hilfreich sein.

Beispiel 63.4 Parameter von einer Konfigurationsdatei laden

```
#include <boost/program_options.hpp>
#include <string>
#include <fstream>
#include <iostream>

using namespace boost::program_options;

int main(int argc, const char *argv[])
{
  try
  {
    options_description generalOptions{"General"};
    generalOptions.add_options()
      ("help,h", "Help screen")
      ("config", value<std::string>(), "Config file");

    options_description fileOptions{"File"};
    fileOptions.add_options()
      ("age", value<int>(), "Age");
```

```
    variables_map vm;
    store(parse_command_line(argc, argv, generalOptions), vm);
    if (vm.count("config"))
    {
      std::ifstream ifs{vm["config"].as<std::string>().c_str()};
      if (ifs)
        store(parse_config_file(ifs, fileOptions), vm);
    }
    notify(vm);

    if (vm.count("help"))
      std::cout << generalOptions << '\n';
    else if (vm.count("age"))
      std::cout << "Your age is: " << vm["age"].as<int>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Beispiel 63.4 verwendet zwei Objekte vom Typ `boost::program_options::options_description`. **`generalOptions`** definiert die Parameter, die auf der Kommandozeile angegeben werden können. **`fileOptions`** legt einen Parameter fest, der von einer Konfigurationsdatei geladen werden kann.
Sie müssen Ihre Parameter nicht zwingend in unterschiedlichen Objekten vom Typ `boost::program_options::options_description` definieren. Eine Aufteilung bietet sich für Beispiel 63.4 insofern an, um zu verhindern, dass `--help` in die Konfigurationsdatei gesetzt wird. Das Programm würde schließlich bei jedem Start die Hilfe anzeigen.
Im Beispiel 63.4 soll `--age` von einer Konfigurationsdatei geladen werden können. Der Name der Konfigurationsdatei soll dabei als Kommandozeilenparameter an das Programm übergeben werden können. Dazu wurde `--config` in **`generalOptions`** definiert.
Nachdem die Kommandozeilenparameter mit `boost::program_options::parse_command_line()` geparst und in **`vm`** gespeichert wurden, wird überprüft, ob `--config` gesetzt ist. Ist dies der Fall, wird die Konfigurationsdatei mit `std::ifstream` geöffnet. Das entsprechende Objekt vom Typ `std::ifstream` wird an eine Funktion `boost::program_options::parse_config_file()` übergeben. Außerdem wird **`fileOptions`** übergeben, das die Parameter beschreibt. `boost::program_options::parse_config_file()` macht das Gleiche wie `boost::program_options::parse_command_line()` und gibt die geparsten Parameter als ein Objekt vom Typ `boost::program_options::parsed_options` zurück. Dieses wird an `boost::program_options::store()` übergeben, um die geparsten Parameter in **`vm`** abzulegen.
Sie können das Programm testen, wenn Sie eine Konfigurationsdatei erstellen und in ihr zum Beispiel age=29 speichern. Heißt die Konfigurationsdatei `config.txt`, rufen Sie das Programm wie folgt auf:

```
test --config config.txt
```

Sie erhalten daraufhin die Ausgabe:

```
Your age is: 29
```

Wenn Sie die gleichen Parameter sowohl auf der Kommandozeile als auch in einer Konfigurationsdatei unterstützen, weil Sie zum Beispiel nur ein Objekt vom Typ `boost::program_options::options_description` verwenden, kann es sein, dass Ihr Programm zweimal den gleichen Parameter parst – einmal mit `boost::program_options::parse_command_line()` und einmal mit `boost::program_options::parse_config_file()`. Es hängt von der Reihenfolge der Funktionsaufrufe ab, welchen Parameterwert Sie in **`vm`** vorfinden. Wenn einmal ein Parameterwert in **`vm`** gespeichert ist, wird er nicht überschrieben. Das heißt, die erste Angabe gilt. Ob diese Angabe von der Kommandozeile oder aus einer Konfigurationsdatei stammt, legen Sie über die Reihenfolge der Funktionsaufrufe fest.
Neben `boost::program_options::parse_command_line()` und `boost::program_options::parse_config_file()` bietet Boost.ProgramOptions auch eine Funktion `boost::program_options::parse_environment()` an, um Parameter von Umgebungsvariablen zu laden. Die Bibliothek stellt auch eine Klasse `boost::environment_iterator` zur Verfügung, mit der über Umgebungsvariablen iteriert werden kann.

Kapitel 64

Boost.Serialization

Die Bibliothek Boost.Serialization ermöglicht es, Objekte in einem C++-Programm in eine Byte-Sequenz umzuwandeln, diese zu speichern und zu einem späteren Zeitpunkt die gleichen Objekte von dieser Byte-Sequenz zu laden. Dabei stehen verschiedene Datenformate einschließlich XML zur Verfügung, die festlegen, nach welchen Regeln die Byte-Sequenz gebildet wird. Alle von Boost.Serialization unterstützten Formate sind proprietär. So kann zum Beispiel das XML-Format nicht genutzt werden, um Daten mit anderen Anwendungen auszutauschen, die nicht in C++ entwickelt sind und nicht Boost.Serialization verwenden. Alle Daten, die im XML-Format gespeichert werden, sind daraufhin ausgerichtet, die gleichen C++-Objekte zu laden, die vorher gespeichert wurden. Das XML-Format hat lediglich den Vorteil, dass Sie die serialisierten C++-Objekte in einem Text-Editor betrachten können, was zum Beispiel die Fehlersuche vereinfachen kann. Sollten Sie auf der Suche nach einer Bibliothek sein, um Daten in standardisierten Formaten mit anderen Programmen auszutauschen, sind Sie bei Boost.Serialization falsch.

64.1 Archive

Das zentrale Konzept in Boost.Serialization ist das *Archiv*. Das Archiv stellt die Byte-Sequenz dar, die serialisierte C++-Objekte repräsentiert. Objekte können einem Archiv hinzugefügt und damit serialisiert werden oder von einem Archiv geladen werden. Dies setzt voraus, dass die gleichen Typen verwendet werden. Nur dann können beim Laden die gleichen C++-Objekte erstellt werden, die zuvor gespeichert wurden.

Beispiel 64.1 `boost::archive::text_oarchive` in Aktion

```
#include <boost/archive/text_oarchive.hpp>
#include <iostream>

using namespace boost::archive;

int main()
{
  text_oarchive oa{std::cout};
  int i = 1;
  oa << i;
}
```

Boost.Serialization stellt mehrere Archiv-Klassen zur Verfügung. So ist in der Headerdatei `boost/archive/text_oarchive.hpp` das Archiv `boost::archive::text_oarchive` definiert. Dieses Archiv ermöglicht es, Objekte als Text-Stream zu serialisieren. So gibt Beispiel 64.1 `22 serialization::archive 11 1` auf die Standardausgabe aus.
Sie können das Objekt **oa** vom Typ `boost::archive::text_oarchive` wie einen Stream verwenden und per `operator<<` eine Variable serialisieren. Sie sollten Archive jedoch nicht als herkömmliche Streams ansehen, in die Sie beliebig Daten ablegen können. Immerhin sollen die in einem Archiv serialisierten Daten später gelesen werden, was erfordert, dass beim Laden exakt die gleichen Typen verwendet werden und Daten in der richtigen Reihenfolge gelesen werden. Sehen Sie sich dazu Beispiel 64.2 an, in dem die Variable vom Typ int nicht nur serialisiert, sondern auch geladen wird.

Beispiel 64.2 `boost::archive::text_iarchive` in Aktion

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <fstream>

using namespace boost::archive;

void save()
{
  std::ofstream file{"archive.txt"};
  text_oarchive oa{file};
  int i = 1;
  oa << i;
}

void load()
{
  std::ifstream file{"archive.txt"};
  text_iarchive ia{file};
  int i = 0;
  ia >> i;
  std::cout << i << '\n';
}

int main()
{
  save();
  load();
}
```

Während die Klasse `boost::archive::text_oarchive` verwendet werden kann, um Daten als Text-Stream zu serialisieren, kann `boost::archive::text_iarchive` verwendet werden, um Daten von einem derartigen Stream wieder zu lesen. Um diese Klasse zu verwenden, muss die Headerdatei `boost/archive/text_iarchive.hpp` eingebunden werden.
Archive erwarten im Konstruktor als Parameter einen Input- oder Output-Stream. Archive greifen auf diese Streams zu, um serialisierte Daten dort auszugeben oder von dort zu laden. So wird im Beispiel 64.2 auf eine Datei zugegriffen. Sie können jedoch auch zum Beispiel einen Stringstream verwenden.

Beispiel 64.3 Serialisierung mit einem Stringstream

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

void save()
{
  text_oarchive oa{ss};
  int i = 1;
  oa << i;
}

void load()
{
  text_iarchive ia{ss};
  int i = 0;
  ia >> i;
  std::cout << i << '\n';
```

```
}

int main()
{
  save();
  load();
}
```

Beispiel 64.3 gibt wie das vorherige `1` auf die Standardausgabe aus. Im Gegensatz zum vorherigen Beispiel serialisiert es Daten nicht in einer Datei, sondern in einem Stringstream.
Sie wissen nun, wie Variablen primitiver Typen serialisiert werden. Im Beispiel 64.4 sehen Sie, wie die Serialisierung von Objekten benutzerdefinierter Typen erfolgt.

Beispiel 64.4 Serialisierung von benutzerdefinierten Typen mit einer Methode

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  text_oarchive oa{ss};
  animal a{4};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
}

int main()
{
  save();
  load();
}
```

Wenn Objekte benutzerdefinierter Typen serialisiert werden sollen, müssen sie wie im Beispiel 64.4 eine Methode namens `serialize()` definieren. Diese Methode wird aufgerufen, wenn ein Objekt serialisiert wird oder von einem Byte-Stream geladen wird. Da `serialize()` sowohl zum Speichern als auch zum Laden verwendet wird,

bietet Boost.Serialization neben `operator<<` und `operator>>` einen Operator an, der automatisch das Richtige tut. Wenn Sie den Operator `operator&` verwenden, müssen Sie in `serialize()` nicht zwischen Speichern und Laden unterscheiden.
Die Methode `serialize()` wird automatisch aufgerufen, wenn ein Objekt serialisiert oder geladen wird. Sie sollte niemals direkt aufgerufen werden und daher privat sein. Damit Boost.Serialization dennoch auf die Methode zugreifen kann, muss die Klasse `boost::serialization::access` als Freund deklariert werden.
Unter Umständen können Sie eine Klasse nicht ändern und ihr keine Methode `serialize()` hinzufügen. Dies trifft zum Beispiel auf Klassen aus der Standardbibliothek zu.

Beispiel 64.5 Serialisierung mit einer freistehenden Funktion

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

struct animal
{
  int legs_;

  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }
};

template <typename Archive>
void serialize(Archive &ar, animal &a, const unsigned int version)
{
  ar & a.legs_;
}

void save()
{
  text_oarchive oa{ss};
  animal a{4};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
}

int main()
{
  save();
  load();
}
```

Wenn Sie einen Typ serialisierbar machen wollen, den Sie nicht ändern können, können Sie wie im Beispiel 64.5 eine freistehende Funktion `serialize()` definieren. Als zweiter Parameter muss dieser Funktion eine Referenz auf ein Objekt des entsprechenden Typs übergeben werden.
`serialize()` als freistehende Funktion implementiert setzt voraus, dass von außerhalb ein Zugriff auf die wesentlichen Eigenschaften einer Klasse möglich ist. So kann `serialize()` im obigen Beispiel nur deswegen als freistehende Funktion implementiert werden, weil die Eigenschaft **`legs_`** in der Klasse `animal` nicht mehr privat ist.
Für viele Klassen aus der Standardbibliothek stellt Boost.Serialization entsprechende `serialize()`-Funktionen

zur Verfügung. Es müssen lediglich zusätzliche Headerdateien eingebunden werden, um Objekte zu serialisieren, die auf Klassen aus der Standardbibliothek basieren.

Beispiel 64.6 Strings serialisieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs, std::string name) :
    legs_{legs}, name_{std::move(name)} {}
  int legs() const { return legs_; }
  const std::string &name() const { return name_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  friend void serialize(Archive &ar, animal &a, const unsigned int version);

  int legs_;
  std::string name_;
};

template <typename Archive>
void serialize(Archive &ar, animal &a, const unsigned int version)
{
  ar & a.legs_;
  ar & a.name_;
}

void save()
{
  text_oarchive oa{ss};
  animal a{4, "cat"};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
  std::cout << a.name() << '\n';
}

int main()
{
  save();
  load();
}
```

Im Beispiel 64.6 wird die Klasse `animal` um einen Namen vom Typ `std::string` erweitert. Damit diese Eigenschaft serialisiert werden kann, muss die Headerdatei `boost/serialization/string.hpp` eingebunden werden. Diese stellt die für `std::string` notwendige freistehende Funktion `serialize()` zur Verfügung. Wie bereits erwähnt, definiert Boost.Serialization für zahlreiche Klassen aus der Standardbibliothek `serialize()`-Funktionen. Diese Funktionen sind in Headerdateien definiert, die den gleichen Namen tragen wie die entsprechenden Headerdateien aus der Standardbibliothek. Wenn Sie wie im Beispiel 64.6 Objekte vom Typ `std::string` serialisieren wollen, müssen Sie die Headerdatei `boost/serialization/string.hpp` einbinden. Wollen Sie ein Objekt vom Typ `std::vector` serialisieren, greifen Sie auf die Headerdatei `boost/serialization/vector.hpp` zu. Es ist also leicht erkennbar, welche Headerdateien Sie aus Boost.Serialization einbinden müssen.
Ein Parameter von `serialize()`, der bisher ignoriert wurde, ist **`version`**. Dieser Parameter ist dann von Bedeutung, wenn Sie davon ausgehen, dass Sie Ihr Programm im Laufe der Zeit weiterentwickeln und es abwärtskompatibel sein soll. So kann Beispiel 64.7 Archive laden, die mit Beispiel 64.5 erstellt wurden. Die Version von `animal` im Beispiel 64.5 enthält keinen Namen. Beispiel 64.7 überprüft beim Laden die Versionsnummer und greift nur dann auf einen Namen zu, wenn die Versionsnummer größer als 0 ist. So können Archive geladen werden, die mit einer älteren Programmversion erstellt wurden.

Beispiel 64.7 Abwärtskompatibilität mit Versionsnummern

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs, std::string name) :
    legs_{legs}, name_{std::move(name)} {}
  int legs() const { return legs_; }
  const std::string &name() const { return name_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  friend void serialize(Archive &ar, animal &a, const unsigned int version);

  int legs_;
  std::string name_;
};

template <typename Archive>
void serialize(Archive &ar, animal &a, const unsigned int version)
{
  ar & a.legs_;
  if (version > 0)
    ar & a.name_;
}

BOOST_CLASS_VERSION(animal, 1)

void save()
{
  text_oarchive oa{ss};
  animal a{4, "cat"};
```

```
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
  std::cout << a.name() << '\n';
}

int main()
{
  save();
  load();
}
```

Das Makro `BOOST_CLASS_VERSION` wird verwendet, um die Versionsnummer für eine Klasse zu setzen. So wird im Beispiel 64.7 die Versionsnummer für die Klasse `animal` auf 1 gesetzt. Standardmäßig – wenn `BOOST_CLASS_VERSION` nicht verwendet wird – ist die Versionsnummer 0.
Die Versionsnummer wird im Archiv gespeichert und ist Bestandteil jedes Archivs. Während beim Speichern die Versionsnummer verwendet wird, die mit `BOOST_CLASS_VERSION` für die entsprechende Klasse angegeben ist, wird beim Laden der Parameter **`version`** der Funktion `serialize()` auf den Wert gesetzt, der im Archiv gespeichert ist. Wenn die neue Version der Klasse `animal` auf ein Archiv zugreifen würde, in dem ein Objekt basierend auf der alten Version dieser Klasse gespeichert ist, würde nicht versucht werden, einen Namen zu lesen. Denn die alte Version der Klasse `animal` besitzt keinen Namen.

64.2 Zeiger und Referenzen

Boost.Serialization kann Zeiger und Referenzen serialisieren. Weil ein Zeiger eine Adresse eines Objekts speichert, ergibt allein die Serialisierung dieser Adresse keinen Sinn. Wenn ein Zeiger oder eine Referenz serialisiert wird, wird automatisch das Objekt serialisiert, auf das der Zeiger oder die Referenz verweist.

Beispiel 64.8 Zeiger serialisieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  boost::archive::text_oarchive oa{ss};
```

```
  animal *a = new animal{4};
  oa << a;
  std::cout << std::hex << a << '\n';
  delete a;
}

void load()
{
  boost::archive::text_iarchive ia{ss};
  animal *a;
  ia >> a;
  std::cout << std::hex << a << '\n';
  std::cout << std::dec << a->legs() << '\n';
  delete a;
}

int main()
{
  save();
  load();
}
```

Im Beispiel 64.8 wird mit `new` ein neues Objekt vom Typ `animal` erstellt und im Zeiger **a** verankert. Dieser Zeiger – und nicht `*a` – wird serialisiert. Dabei speichert Boost.Serialization nicht die Adresse des Objekts, die im Zeiger gespeichert ist. Es wird automatisch das Objekt serialisiert, auf das **a** zeigt.
Beim Laden des Archivs wird **a** nicht notwendigerweise auf die gleiche Adresse gesetzt. Stattdessen wird automatisch ein neues Objekt erstellt, dessen Adresse in **a** gespeichert wird. Boost.Serialization garantiert, dass das Objekt dem gleicht, das zuvor serialisiert wurde. Für die Adresse gilt dies nicht.
Da im Zusammenhang mit dynamisch reserviertem Speicher Smartpointer verwendet werden, bietet Boost.Serialization auch für diese eine entsprechende Unterstützung an.

Beispiel 64.9 Smartpointer serialisieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/scoped_ptr.hpp>
#include <boost/scoped_ptr.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  text_oarchive oa{ss};
  boost::scoped_ptr<animal> a{new animal{4}};
  oa << a;
```

```
}

void load()
{
  text_iarchive ia{ss};
  boost::scoped_ptr<animal> a;
  ia >> a;
  std::cout << a->legs() << '\n';
}

int main()
{
  save();
  load();
}
```

Im Beispiel 64.9 wird der Smartpointer `boost::scoped_ptr` verwendet, um ein dynamisch reserviertes Objekt vom Typ `animal` zu verwalten. Damit ein Zeiger vom Typ `boost::scoped_ptr` serialisiert werden kann, muss die Headerdatei `boost/serialization/scoped_ptr.hpp` eingebunden werden.
Für den Fall, dass Sie den Smartpointer `boost::shared_ptr` serialisieren möchten, müssen Sie die Headerdatei `boost/serialization/shared_ptr.hpp` verwenden. Diese Headerdatei unterstützt auch die Serialisierung des Smartpointers `std::shared_ptr`.
Beispiel 64.10 verwendet anstatt eines Zeigers eine Referenz.

Beispiel 64.10 Referenzen serialisieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  text_oarchive oa{ss};
  animal a{4};
  animal &r = a;
  oa << r;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  animal &r = a;
  ia >> r;
```

```
  std::cout << r.legs() << '\n';
}

int main()
{
  save();
  load();
}
```

Ähnlich wie bei Zeigern wird auch hier automatisch das referenzierte Objekt serialisiert.

64.3 Serialisieren von Objekten aus Klassenhierarchien

Wenn Objekte serialisiert werden sollen, die auf Typen aus Klassenhierarchien basieren, muss in Kindklassen innerhalb der Methode `serialize()` auf eine Funktion `boost::serialization::base_object()` zugegriffen werden. Nur diese Funktion stellt sicher, dass von Elternklassen geerbte Eigenschaften einwandfrei serialisiert werden. Sehen Sie sich dazu Beispiel 64.11 an, das um eine Klasse `bird` erweitert wurde, die von `animal` abgeleitet ist.

Beispiel 64.11 Kindklassen richtig serialisieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;
std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

class bird : public animal
{
public:
  bird() = default;
  bird(int legs, bool can_fly) :
    animal{legs}, can_fly_{can_fly} {}
  bool can_fly() const { return can_fly_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<animal>(*this);
    ar & can_fly_;
```

```
  }

  bool can_fly_;
};

void save()
{
  text_oarchive oa{ss};
  bird penguin{2, false};
  oa << penguin;
}

void load()
{
  text_iarchive ia{ss};
  bird penguin;
  ia >> penguin;
  std::cout << penguin.legs() << '\n';
  std::cout << std::boolalpha << penguin.can_fly() << '\n';
}

int main()
{
  save();
  load();
}
```

Beide Klassen – sowohl `animal` als auch `bird` – besitzen eine private Methode `serialize()`. Demnach können Objekte, die auf einer dieser Klassen basieren, serialisiert werden. Weil `bird` jedoch von `animal` abgeleitet ist, muss die Methode `serialize()` in dieser Klasse dafür sorgen, dass die von `animal` geerbten Eigenschaften ebenfalls serialisiert werden.
Geerbte Eigenschaften werden serialisiert, indem in der Kindklasse in `serialize()` mit `boost::serialization::base_object()` auf die Elternklasse zugegriffen wird. Es ist zwingend notwendig, diese Funktion und nicht zum Beispiel `static_cast` zu verwenden, weil nur `boost::serialization::base_object()` eine einwandfreie Serialisierung sicherstellt.
Wenn Objekte dynamisch erzeugt werden, können ihre Adressen in Zeigern vom Typ der Elternklasse gespeichert werden. Dass Boost.Serialization auch in diesem Fall Objekte korrekt serialisieren kann, zeigt Beispiel 64.12.

Beispiel 64.12 Kindklassen statisch mit `BOOST_CLASS_EXPORT` registrieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/export.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  virtual int legs() const { return legs_; }
  virtual ~animal() = default;

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }
```

```
  int legs_;
};

class bird : public animal
{
public:
  bird() = default;
  bird(int legs, bool can_fly) :
    animal{legs}, can_fly_{can_fly} {}
  bool can_fly() const { return can_fly_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<animal>(*this);
    ar & can_fly_;
  }

  bool can_fly_;
};

BOOST_CLASS_EXPORT(bird)

void save()
{
  text_oarchive oa{ss};
  animal *a = new bird{2, false};
  oa << a;
  delete a;
}

void load()
{
  text_iarchive ia{ss};
  animal *a;
  ia >> a;
  std::cout << a->legs() << '\n';
  delete a;
}

int main()
{
  save();
  load();
}
```

Im Beispiel 64.12 wird in der Funktion `save()` ein Objekt vom Typ `bird` erstellt und in einem Zeiger vom Typ animal* verankert. Dieser Zeiger wird per `operator<<` an das Archiv übergeben.
Wie bereits im vorherigen Abschnitt gesehen, wird in diesem Fall automatisch das Objekt serialisiert, auf das der Zeiger verweist. Damit Boost.Serialization erkennt, dass ein Objekt vom Typ `bird` serialisiert werden muss, obwohl der Zeiger den Typ animal* hat, muss die Klasse `bird` der Bibliothek bekannt gemacht werden. Dies erfolgt mit Hilfe des Makros `BOOST_CLASS_EXPORT`, das in der Headerdatei `boost/serialization/export.hpp` definiert ist. Ohne dieses Makro könnte Boost.Serialization das Objekt vom Typ `bird` nicht richtig serialisieren, weil der Typ `bird` nicht in der Zeigerdefinition auftaucht und daher für Boost.Serialization unbekannt wäre.
Sie müssen das Makro `BOOST_CLASS_EXPORT` immer dann verwenden, wenn Sie Kindklassen erstellen und Objekte vom Typ dieser Kindklassen serialisieren wollen, dies aber über einen Zeiger vom Typ einer Elternklasse tun.
Ein Nachteil des Makros `BOOST_CLASS_EXPORT` ist, dass aufgrund der statischen Registrierung auch Klassen re-

gistiert werden, die womöglich in einem Programm nicht zur Serialisierung verwendet werden. Boost.Serialization bietet auch für diesen Fall eine Lösung an.

Beispiel 64.13 Kindklassen dynamisch mit `register_type()` registrieren

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/export.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  virtual int legs() const { return legs_; }
  virtual ~animal() = default;

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

class bird : public animal
{
public:
  bird() = default;
  bird(int legs, bool can_fly) :
    animal{legs}, can_fly_{can_fly} {}
  bool can_fly() const { return can_fly_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<animal>(*this);
    ar & can_fly_;
  }

  bool can_fly_;
};

void save()
{
  boost::archive::text_oarchive oa{ss};
  oa.register_type<bird>();
  animal *a = new bird{2, false};
  oa << a;
  delete a;
}

void load()
{
  boost::archive::text_iarchive ia{ss};
  ia.register_type<bird>();
  animal *a;
```

```
  ia >> a;
  std::cout << a->legs() << '\n';
  delete a;
}

int main()
{
  save();
  load();
}
```

Anstatt das Makro `BOOST_CLASS_EXPORT` zu verwenden, wird im Beispiel 64.13 für ein Archiv die Methode `register_type()` aufgerufen. Ihr wird der entsprechende Typ, der registriert werden soll, als Template-Parameter übergeben.
Beachten Sie, dass `register_type()` sowohl in `save()` als auch in `load()` aufgerufen werden muss.
Der Vorteil der dynamischen Registrierung mit `register_type()` ist, dass zur Laufzeit entschieden werden kann, welche Typen für die Serialisierung verwendet und daher registriert werden müssen.

64.4 Wrapper-Funktionen zur Optimierung

Während Sie Boost.Serialization soweit kennengelernt haben, dass Sie Objekte serialisieren können, werden Ihnen abschließend Wrapper-Funktionen vorgestellt, mit denen Sie die Serialisierung optimieren können. Indem Sie Wrapper-Funktionen verwenden, markieren Sie gewissermaßen Objekte, was Boost.Serialization ermöglicht, entsprechende Optimierungstechniken anzuwenden.

Beispiel 64.14 Serialisierung eines Arrays auf herkömmliche Weise

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/array.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

void save()
{
  text_oarchive oa{ss};
  boost::array<int, 3> a{{0, 1, 2}};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  boost::array<int, 3> a;
  ia >> a;
  std::cout << a[0] << ", " << a[1] << ", " << a[2] << '\n';
}

int main()
{
  save();
  load();
}
```

Der Text-Stream, der im Beispiel 64.14 bei der Serialisierung erstellt wird, ist `22 serialization::archive 11 0 0 3 0 1 2`. Mit Hilfe der im Beispiel 64.15 verwendeten Wrapper-Funktion `boost::serialization::make_array()` kann der Text-Stream auf `22 serialization::archive 11 0 1 2` verkürzt werden.

Beispiel 64.15 Serialisierung eines Arrays mit `make_array()`

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/array.hpp>
#include <array>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

void save()
{
  text_oarchive oa{ss};
  std::array<int, 3> a{{0, 1, 2}};
  oa << boost::serialization::make_array(a.data(), a.size());
}

void load()
{
  text_iarchive ia{ss};
  std::array<int, 3> a;
  ia >> boost::serialization::make_array(a.data(), a.size());
  std::cout << a[0] << ", " << a[1] << ", " << a[2] << '\n';
}

int main()
{
  save();
  load();
}
```

Der Funktion `boost::serialization::make_array()` muss die Adresse eines Arrays und dessen Länge übergeben werden. Weil die Länge im Voraus bekannt ist, muss sie nicht als Bestandteil des Objekts vom Typ `boost::array` serialisiert werden.
`boost::serialization::make_array()` bietet sich immer dann an, wenn eine Klasse wie `std::array` oder `std::vector` ein Array enthalten und das Array direkt serialisiert werden kann. Andere Eigenschaften, die womöglich zur Serialisierung der entsprechenden Klasse gespeichert werden müssten, werden nicht serialisiert.
Eine weitere Wrapper-Funktion, die Boost.Serialization anbietet, ist `boost::serialization::make_binary_object()`. Ihr muss ähnlich wie `boost::serialization::make_array()` eine Adresse und eine Länge übergeben werden. Der Unterschied zwischen den beiden Funktionen ist, dass `boost::serialization::make_binary_object()` ausschließlich für Binärdaten verwendet wird, denen keinerlei Struktur zugrunde liegt, während `boost::serialization::make_array()` für Arrays verwendet wird.

Kapitel 65

Boost.Uuid

Die Bibliothek Boost.Uuid bietet Generatoren an, die *UUIDs* erzeugen können. Dabei handelt es sich um IDs, die weltweit eindeutig sind, ohne dass eine zentrale koordinierende Instanz notwendig ist. Es gibt also zum Beispiel keine Datenbank, die weltweit alle UUIDs speichert, um sicherzustellen, dass Sie eine neue noch nicht benutzte ID erhalten.
UUIDs werden in dezentral entwickelten Systemen verwendet, in denen Komponenten eindeutig identifiziert werden müssen. Zum Beispiel verwendet die von Microsoft entwickelte COM-Technologie UUIDs, um Schnittstellen zu identifizieren. Entwickeln Microsoft oder andere neue COM-Schnittstellen, können diesen schnell und einfach eindeutige IDs zugewiesen werden.
Bei UUIDs handelt es sich um 128-Bit-Zahlen. Es gibt verschiedene Möglichkeiten sicherzustellen, dass generierte UUIDs weltweit eindeutig sind. So kann beispielsweise die Netzwerkadresse eines Computers bei der Generierung einbezogen werden. Die von Boost.Uuid verwendeten Generatoren basieren auf einem Zufallsmechanismus, um zu verhindern, dass UUIDs Daten enthalten, die verraten, auf welchem Computer sie erzeugt wurden.
Alle von Boost.UUID angebotenen Klassen und Funktionen sind im Namensraum `boost::uuids` definiert. Es gibt jedoch keine Master-Headerdatei, die Sie einbinden können, um Zugriff auf alle Klassen und Funktionen von Boost.Uuid zu erhalten.

Beispiel 65.1 Zufällige UUIDs generieren mit `boost::uuids::random_generator`

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();
  std::cout << id << '\n';
}
```

Im Beispiel 65.1 wird eine zufällige UUID generiert. Dazu wird auf die Klasse `boost::uuids::random_generator` zugegriffen. Um diesen Generator verwenden zu können, muss die Headerdatei `boost/uuid/random_generator.hpp` eingebunden werden. Über die im Beispiel 65.1 verwendete Headerdatei `boost/uuid/uuid_generators.hpp` kann auf alle von Boost.Uuid angebotenen Generatoren zugegriffen werden.
`boost::uuids::random_generator` wird wie die Generatoren aus der C++11-Standardbibliothek oder aus Boost.Random verwendet. Die Klasse überlädt den Operator `operator()`, auf den Sie zugreifen, um eine neue zufällige UUID zu erzeugen.
UUIDs haben den Typ `boost::uuids::uuid`. Es handelt sich hierbei um eine *POD*-Struktur – auf Englisch plain old data. Sie können Objekte vom Typ `boost::uuids::uuid` nicht ohne Generator initialisieren. Dafür handelt es sich um einen schlanken Typ, der im Speicher genau 128 Bits belegt. `boost::uuids::uuid` ist in der Headerdatei `boost/uuid/uuid.hpp` definiert, die deswegen im Beispiel 65.1 eingebunden wird.
Ein Objekt vom Typ `boost::uuids::uuid` kann direkt auf die Standardausgabe ausgegeben werden. Dazu muss jedoch die Headerdatei `boost/uuid/uuid_io.hpp` eingebunden werden. In dieser Headerdatei befinden sich die überladenen Operatoren, damit Objekte vom Typ `boost::uuids::uuid` an Streams übergeben

werden können.
Wenn Sie Beispiel 65.1 ausführen, wird Ihnen zum Beispiel `0cb6f61f-be68-5afc-8686-c52e3fc7a50d` ausgegeben. Die Einteilung in Blöcke, die mit Bindestrichen voneinander getrennt sind, ist die bevorzugte Darstellung von UUIDs.

Beispiel 65.2 Methoden von `boost::uuids::uuid`

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();
  std::cout << id.size() << '\n';
  std::cout << std::boolalpha << id.is_nil() << '\n';
  std::cout << id.variant() << '\n';
  std::cout << id.version() << '\n';
}
```

`boost::uuids::uuid` bietet nur wenige Methoden an, von denen Ihnen einige im Beispiel 65.2 vorgestellt werden. `size()` gibt die Größe einer UUID in Bytes zurück. Da eine UUID per Definition 128 Bit groß ist, gibt `size()` immer 16 zurück. `is_nil()` gibt `true` zurück, wenn es sich bei einer UUID um eine Null-UUID handelt. Die Null-UUID ist 00000000-0000-0000-0000-000000000000. `variant()` und `version()` geben an, um welche Variante es sich bei einer UUID handelt und mit welchem Verfahren sie generiert wurde. Für Beispiel 65.2 gibt `variant()` 1 zurück, was bedeutet, dass sich die UUID nach der RFC-Spezifikation 4122 richtet. `version()` gibt 4 zurück. Das bedeutet, dass die UUID von einem Zufallsgenerator erzeugt wurde.
`boost::uuids::uuid` bietet außerdem Methoden wie `begin()`, `end()` und `swap()` an.

Beispiel 65.3 Generatoren von Boost.Uuid

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  nil_generator nil_gen;
  uuid id = nil_gen();
  std::cout << std::boolalpha << id.is_nil() << '\n';

  string_generator string_gen;
  id = string_gen("CF77C981-F61B-7817-10FF-D916FCC3EAA4");
  std::cout << id.variant() << '\n';

  name_generator name_gen(id);
  std::cout << name_gen("theboostcpplibraries.com") << '\n';
}
```

Im Beispiel 65.3 sehen Sie weitere von Boost.Uuid angebotene Generatoren. Mit `nil_generator` kann eine Null-UUID generiert werden. Nur für eine Null-UUID gibt `is_nil()` `true` zurück.
`string_generator` wird verwendet, wenn Sie bereits eine UUID haben und sie in Ihrem Programm verwenden wollen. So können Sie zum Beispiel UUIDs auf `http://www.uuidgenerator.net/` generieren. Für die im Beispiel verwendete UUID gibt `variant()` 0 zurück, was bedeutet, dass die UUID dem abwärtskompatiblen Standard NCS entspricht. Die Klasse `name_generator` wiederum wird verwendet, um UUIDs in Namensräumen zu generieren.
Achten Sie auf die Schreibweise von UUIDs, wenn Sie die Klasse `string_generator` verwenden. Die Bindestriche müssen an den richtigen Stellen gesetzt sein. Alternativ können Sie UUIDs ohne Bindestriche angeben. Groß- und Kleinschreibung spielt keine Rolle.

Beispiel 65.4 Einfache Umwandlung in Strings

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();

  std::string s = to_string(id);
  std::cout << s << '\n';

  std::cout << boost::lexical_cast<std::string>(id) << '\n';
}
```

Boost.Uuid bietet wie im Beispiel 65.4 zu sehen freistehende Funktionen wie `boost::uuids::to_string()` und `boost::uuids::to_wstring()` an, um UUIDs ohne den Umweg über Streams in einen String umwandeln zu können. Es ist außerdem möglich, UUIDs mit `boost::lexical_cast()` in einen String umzuwandeln.

Teil XVI

Entwurfsmuster

Mit Boost.Flyweight, Boost.Signals2 und Boost.MetaStateMachine werden drei Boost-Bibliotheken für Entwurfsmuster vorgestellt.

- Boost.Flyweight hilft Ihnen, wenn Sie viele gleiche Objekte in Ihrem Programm verwenden und den Speicherbedarf einschränken wollen. Das entsprechende Entwurfsmuster heißt auf Deutsch Fliegengewicht.
- Boost.Signals2 macht es einfach, das Beobachter-Muster einzusetzen. Die Bibliothek heißt Boost.Signals2, weil sie das Signal-Slot-Konzept umsetzt.
- Mit Boost.MetaStateMachine können Sie einen Zustandsautomaten aus der UML in C++-Code übertragen.

Kapitel 66

Boost.Flyweight

Boost.Flyweight ist eine Bibliothek, die es einfach macht, das gleichnamige Entwurfsmuster einzusetzen – auf Deutsch Fliegengewicht genannt. Dabei geht es darum, Speicherplatz zu sparen, wenn sich viele Objekte Daten teilen. Anstatt die gleichen Daten mehrfach in Objekten zu speichern, sorgt das Entwurfsmuster Flyweight dafür, dass die geteilten Daten nur einmal im Speicher gehalten werden und alle Objekte auf diese Daten verweisen. Während man dieses Entwurfsmuster auch selbst zum Beispiel über Zeiger umsetzen könnte, geht dies mit einer Bibliothek wie Boost.Flyweight einfacher.

Beispiel 66.1 Hunderttausend gleiche Strings ohne Boost.Flyweight

```
#include <string>
#include <vector>

struct person
{
  int id_;
  std::string city_;
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Im Beispiel 66.1 werden hunderttausend Objekte vom Typ `person` erstellt. `person` besitzt zwei Eigenschaften: Über **`id`**_ werden Personen identifiziert, und **`city`**_ speichert die Stadt, in der Personen leben. In diesem Beispiel leben alle Personen in Berlin. Deswegen wird die Eigenschaft **`city`**_ aller hunderttausend Objekte auf „Berlin“ gesetzt. Im Beispiel werden folglich hunderttausend Strings verwendet, die alle den gleichen Wert speichern. Mit Boost.Flyweight kann anstatt hunderttausend Strings ein String verwendet und der Speicherbedarf drastisch reduziert werden.

Beispiel 66.2 Ein String statt hunderttausend Strings mit Boost.Flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
```

```
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Um Boost.Flyweight zu verwenden, muss wie im Beispiel 66.2 die Headerdatei `boost/flyweight.hpp` eingebunden werden. Boost.Flyweight bietet weitere Headerdateien an, die jedoch nur eingebunden werden müssen, wenn detaillierte Einstellungen in der Bibliothek geändert werden sollen.
Alle von Boost.Flyweight definierten Klassen und Funktionen befinden sich im Namensraum `boost::flyweights`. Im Beispiel 66.2 wird ausschließlich auf die Klasse `boost::flyweights::flyweight` zugegriffen, die die wichtigste Klasse dieser Bibliothek darstellt. Die Eigenschaft **`city_`** hat nun nicht mehr den Typ `std::string`, sondern den Typ flyweight<std::string>. Diese Änderung reicht aus, um das Fliegengewicht-Entwurfsmuster einzusetzen und den Speicherbedarf des Programms zu verringern.

Beispiel 66.3 `boost::flyweights::flyweight` mehrfach verwenden

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  flyweight<std::string> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Im Beispiel 66.3 wurde der Klasse `person` eine zweite Eigenschaft **`country_`** hinzugefügt. Diese Eigenschaft soll speichern, in welchem Land Personen leben. Da alle Personen in Berlin leben, leben sie folglich auch im gleichen Land. Deswegen wird `boost::flyweights::flyweight` auch zur Definition der Eigenschaft **`country_`** verwendet.
Boost.Flyweight verwendet intern einen Container, in dem Objekte gespeichert werden. Er stellt sicher, dass nicht mehrere Objekte mit gleichen Werten existieren. Boost.Flyweight verwendet dazu standardmäßig einen Hash-Container – also einen Container ähnlich wie `std::unordered_set`. Für unterschiedliche Typen werden unterschiedliche Hash-Container verwendet. Da im Beispiel 66.3 beide Eigenschaften **`city_`** und **`country_`** Strings sind, wird nur ein Container verwendet. Das ist in diesem Beispiel kein Problem, da der Container nur zwei Strings mit „Berlin“ und „Germany“ speichert. Würden viele unterschiedliche Städte und Länder gespeichert werden müssen, wäre es besser, wenn Städte in einem und Länder in einem anderen Container gespeichert werden würden.

Beispiel 66.4 `boost::flyweights::flyweight` mit Tags mehrfach verwenden

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct city {};
struct country {};
```

```
struct person
{
  int id_;
  flyweight<std::string, tag<city>> city_;
  flyweight<std::string, tag<country>> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Im Beispiel 66.4 wird ein zweiter Template-Parameter an `boost::flyweights::flyweight` übergeben. Es handelt sich hierbei um einen *Tag*. Tags sind beliebige Typen, die ausschließlich dazu dienen, die Typen, auf denen **`city`**_ und **`country`**_ basieren, unterscheidbar zu machen. So sind im Beispiel 66.4 zwei leere Strukturen `city` und `country` definiert worden, die als Tags verwendet werden. Es hätten aber genauso gut zum Beispiel int und bool verwendet werden können.
Die Tags führen dazu, dass **`city`**_ und **`country`**_ auf unterschiedlichen Typen basieren. Somit werden zwei Hash-Container von Boost.Flyweight verwendet – der eine speichert Städte, der andere Länder.

Beispiel 66.5 Template-Parameter von `boost::flyweights::flyweight`

```
#include <boost/flyweight.hpp>
#include <boost/flyweight/set_factory.hpp>
#include <boost/flyweight/no_locking.hpp>
#include <boost/flyweight/no_tracking.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string, set_factory<>, no_locking, no_tracking> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Außer einem Tag können `boost::flyweights::flyweight` auch andere Template-Parameter übergeben werden. Im Beispiel 66.5 sind dies `boost::flyweights::set_factory`, `boost::flyweights::no_locking` und `boost::flyweights::no_tracking`. Beachten Sie, dass zur Verwendung dieser Klassen zusätzliche Headerdateien eingebunden werden müssen.
`boost::flyweights::set_factory` gibt an, dass Boost.Flyweight keinen Hash-Container, sondern einen sortierten Container ähnlich wie `std::set` verwenden soll. Über `boost::flyweights::no_locking` wird die Unterstützung für Multithreading deaktiviert, die standardmäßig verwendet wird. `boost::flyweights::no_tracking` wiederum gibt an, dass Boost.Flyweight nicht erfassen soll, ob Objekte, die in den intern verwendeten Containern gespeichert sind, verwendet werden. Werden Objekte nicht mehr verwendet, erkennt Boost.Flyweight das standardmäßig und kann sie von den intern verwendeten Containern entfernen. Mit `boost::flyweights::no_tracking` wird dieser Erkennungsmechanismus ausgeschaltet. Das führt zu einer höheren Performance, bedeutet aber auch, dass intern verwendete Container nur größer werden können und nie kleiner.

Boost.Flyweight bietet weitergehende Einstellungsmöglichkeiten an. So kann zum Beispiel im Detail angegeben werden, welcher Container intern verwendet werden soll.

Kapitel 67

Boost.Signals2

Boost.Signals2 setzt das Signal-Slot-Konzept um. Dabei werden ein oder mehrere Funktionen – Slots genannt – mit einem Objekt verknüpft, das ein Signal aussenden kann. Jedes Mal, wenn das Signal ausgesendet wird, werden alle verknüpften Funktionen aufgerufen.
Das Signal-Slot-Konzept ist zum Beispiel in der Entwicklung von Anwendungen mit grafischen Benutzeroberflächen sinnvoll. Dort können Schaltflächen, die auf Benutzereingaben reagieren sollen, so modelliert sein, dass sie bei einem Klick ein Signal aussenden. Um auf Benutzereingaben reagieren zu können, könnten sie Verknüpfungen mit beliebig vielen Funktionen unterstützen. So könnte flexibel auf Ereignisse reagiert werden.
Sie können auch `std::function` zur Ereignisbehandlung verwenden. Ein entscheidender Unterschied zwischen `std::function` und Boost.Signals2 ist, dass mit `std::function` nur eine Funktion verknüpft werden kann, während Boost.Signals2 die Verknüpfung eines Signals mit beliebig vielen Funktionen erlaubt. Boost.Signals2 unterstützt eine ereignisorientierte Programmierung wesentlich besser und sollte die erste Wahl sein, wenn Ereignisse verarbeitet werden müssen.
Boost.Signals2 ist der Nachfolger der Bibliothek Boost.Signals, die veraltet ist und in diesem Buch nicht vorgestellt wird.

67.1 Signale

Boost.Signals2 bietet eine Klasse `boost::signals2::signal` an, mit der Sie ein Signal erstellen. Die Klasse ist in der Headerdatei `boost/signals2/signal.hpp` definiert. Da mit `boost/signals2.hpp` eine Headerdatei zur Verfügung steht, die alle in Boost.Signals2 definierten Klassen und Funktionen verfügbar macht, reicht es, wenn Sie diese Headerdatei einbinden.
Die Klasse `signal` befindet sich im Namensraum `boost::signals2`. Alle Klassen und Funktionen von Boost.Signals2 sind ohne Ausnahme in diesem Namensraum definiert.

Beispiel 67.1 „Hello, world!“ mit `boost::signals2::signal`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
}
```

Die Klasse `boost::signals2::signal` ist ein Template und erwartet die Signatur der Funktionen, die als Ereignisverarbeiter verwendet werden, als Template-Parameter. Im Beispiel 67.1 können mit dem Signal **s** nur Funktionen verknüpft werden, deren Signatur zu void() kompatibel ist.
Über die Methode `connect()` wird eine Lambda-Funktion mit dem Signal **s** verknüpft. Das funktioniert, weil die Lambda-Funktion kompatibel zur Signatur void() ist. Die Lambda-Funktion ist ein Ereignisverarbeiter, der aufgerufen wird, wenn das Signal **s** ausgelöst wird.

Anschließend wird auf **s** so zugegriffen, als würde es sich um eine Funktion handeln. Die Signatur entspricht dabei dem Template-Parameter: Die runden Klammern bleiben leer. Es werden keine Parameter übergeben, weil die Signatur void() keine Parameter erwartet.
Durch den Aufruf von **s** wird ein Signal ausgelöst, das zum Aufruf der Lambda-Funktion führt, weil diese Funktion mit `connect()` mit dem Signal verknüpft ist.
Beispiel 67.1 kann genauso gut mit `std::function` umgesetzt werden.

Beispiel 67.2 "Hello, world!" mit `std::function`

```
#include <functional>
#include <iostream>

int main()
{
  std::function<void()> f;
  f = []{ std::cout << "Hello, world!\n"; };
  f();
}
```

Im Beispiel 67.2 wird bei einem Aufruf von **f** die verknüpfte Lambda-Funktion aufgerufen. Während mit `std::function` nicht viel mehr möglich ist, können wie im Beispiel 67.3 mehrere Ereignisverarbeiter mit einem Signal vom Typ `boost::signals2::signal` verknüpft werden.

Beispiel 67.3 Mehrere Ereignisverarbeiter mit `boost::signals2::signal`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello"; });
  s.connect([]{ std::cout << ", world!\n"; });
  s();
}
```

Sie können für ein Objekt vom Typ `boost::signals2::signal` die Methode `connect()` mehrfach aufrufen und mehrere Funktionen mit einem Signal verknüpfen. Wird das Signal ausgelöst, werden die Funktionen in der Reihenfolge ausgeführt, in der sie mit `connect()` mit dem Signal verknüpft wurden.
Die Methode `connect()` ist überladen, so dass Sie alternativ explizit angeben können, in welcher Reihenfolge Funktionen aufgerufen werden sollen. Dazu wird `connect()` als zusätzlicher Parameter ein int-Wert übergeben.

Beispiel 67.4 Ereignisverarbeiter mit expliziter Reihenfolge

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect(1, []{ std::cout << ", world!\n"; });
  s.connect(0, []{ std::cout << "Hello"; });
  s();
}
```

Beispiel 67.4 gibt wie das vorherige `Hello, world!` auf die Standardausgabe aus.
Es ist nicht nur möglich, mit `connect()` eine Funktion mit einem Signal zu verbinden, sondern diese Verbindung mit `disconnect()` wieder zu lösen.

Beispiel 67.5 Ereignisverarbeiter von `boost::signals2::signal` lösen

```
#include <boost/signals2.hpp>
#include <iostream>
```

```
using namespace boost::signals2;

void hello() { std::cout << "Hello"; }
void world() { std::cout << ", world!\n"; }

int main()
{
  signal<void()> s;
  s.connect(hello);
  s.connect(world);
  s.disconnect(world);
  s();
}
```

Beispiel 67.5 gibt lediglich `Hello` aus, weil vor Aussenden des Signals die Verknüpfung mit der Funktion `world()` gelöst wurde.
Neben `connect()` und `disconnect()` bietet die Klasse `boost::signals2::signal` nur wenige weitere Methoden an. Sehen Sie sich dazu Beispiel 67.6 an.

Beispiel 67.6 Verschiedene von `boost::signals2::signal` angebotene Methoden

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello"; });
  s.connect([]{ std::cout << ", world!"; });
  std::cout << s.num_slots() << '\n';
  if (!s.empty())
    s();
  s.disconnect_all_slots();
}
```

`num_slots()` gibt die Anzahl der verknüpften Funktionen zurück. Soll überprüft werden, ob keine Funktionen verknüpft sind, kann `empty()` aufgerufen werden. Die Methode `disconnect_all_slots()` wiederum löst alle verknüpften Funktionen vom Signal.

Beispiel 67.7 Rückgabewert von Ereignisverarbeitern auswerten

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<int()> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::cout << *s() << '\n';
}
```

Im Beispiel 67.7 sind zwei Lambda-Funktionen mit einem Signal **s** verknüpft. Die erste Lambda-Funktion gibt 1 zurück, die zweite 2.
Wenn Sie das Beispiel ausführen, wird `2` auf die Standardausgabe ausgegeben. Die Rückgabewerte der Lambda-Funktionen werden von **s** entgegengenommen, bis auf den letzten Rückgabewert aber ignoriert. Standardmäßig wird lediglich der letzte Rückgabewert aller aufgerufenen Funktionen von einem Signal zurückgegeben.
Beachten Sie, dass `s()` den Rückgabewert der letzten aufgerufenen Funktion nicht direkt zurückgibt. Stattdessen wird ein Objekt vom Typ `boost::optional` zurückgegeben, das dereferenziert werden muss, um die Zahl 2 zu erhalten. Boost.Signals2 greift auf `boost::optional` zu, weil beim Auslösen eines Signals, mit dem keine

Funktionen verknüpft sind, keine Rückgabewerte zur Verfügung stehen. In diesem Fall wird ein leeres Objekt vom Typ `boost::optional` zurückgegeben. `boost::optional` wird im Kapitel 21 vorgestellt.
Es ist möglich, ein Signal so anzupassen, dass Rückgabewerte anderweitig verarbeitet werden. Dazu muss der Klasse `boost::signals2::signal` als zweiter Template-Parameter ein *Combiner* übergeben werden.

Beispiel 67.8 Kleinsten Rückgabewert mit benutzerdefiniertem Combiner finden

```
#include <boost/signals2.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::signals2;

template <typename T>
struct min_element
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    std::vector<T> v(first, last);
    return *std::min_element(v.begin(), v.end());
  }
};

int main()
{
  signal<int(), min_element<int>> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::cout << s() << '\n';
}
```

Ein Combiner ist eine Klasse, die den Operator `operator()` überlädt. Diese Methode wird automatisch aufgerufen und bekommt zwei Iteratoren übergeben, mit denen auf die mit dem Signal verbundenen Funktionen zugegriffen werden kann. Diese werden beim Dereferenzieren der Iteratoren aufgerufen, so dass ihre Rückgabewerte im Combiner zur Verfügung stehen. Dort kann beispielsweise wie im Beispiel 67.8 ein Algorithmus aus der Standardbibliothek verwendet werden, um den kleinsten Wert zu ermitteln und zurückzugeben.
Der Combiner, der von `boost::signals2::signal` standardmäßig verwendet wird, ist `boost::signals2::optional_last_value`. Dieser Combiner gibt wie bereits erläutert ein Objekt vom Typ `boost::optional` zurück, das leer ist, wenn ein Signal ausgelöst wird, mit dem keine Funktion verknüpft ist. Wird ein Combiner selbst definiert, kann für den Rückgabewert ein beliebiger Typ verwendet werden. So verwendet der Combiner `min_element` im Beispiel 67.8 als Typ des Rückgabewerts den Template-Parameter, der bei der Instanziierung in `main()` an `min_element` übergeben wird.
Es ist nicht möglich, einen Algorithmus wie `std::min_element()` direkt als Template-Parameter an `boost::signals2::signal` zu übergeben. Die Klasse `boost::signals2::signal` erwartet, dass der Combiner einen Typ result_type definiert. Dieser Typ beschreibt den Rückgabewert von `operator()`. Fehlt er wie bei Algorithmen aus der Standardbibliothek, meldet der Compiler einen Fehler.
Beachten Sie, dass es außerdem im Beispiel 67.8 nicht möglich ist, die Iteratoren **`first`** und **`last`** direkt an `std::min_element()` zu übergeben. Dieser Algorithmus erwartet Forward-Iteratoren, während Combiner mit Input-Iteratoren arbeiten. Das Beispiel nimmt deswegen einen Umweg über einen Vektor, der alle Rückgabewerte speichert, bevor der kleinste Wert mit `std::min_element()` ermittelt wird.
Im Beispiel 67.9 wird der Combiner so geändert, dass Rückgabewerte nicht ausgewertet werden, sondern in einem Container zurückgegeben werden.

Beispiel 67.9 Alle Rückgabewerte mit benutzerdefiniertem Combiner erhalten

```
#include <boost/signals2.hpp>
#include <vector>
#include <algorithm>
#include <iostream>
```

```
using namespace boost::signals2;

template <typename T>
struct return_all
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    return T(first, last);
  }
};

int main()
{
  signal<int(), return_all<std::vector<int>>> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::vector<int> v = s();
  std::cout << *std::min_element(v.begin(), v.end()) << '\n';
}
```

67.2 Verbindungen

Sie können Funktionen mit Hilfe der Methoden `connect()` und `disconnect()`, die von `boost::signals2::signal` angeboten werden, verwalten. Da `connect()` einen Rückgabewert vom Typ `boost::signals2::connection` besitzt, geht dies auch anderweitig.

Beispiel 67.10 Verküpfungen mit `boost::signals2::connection` verwalten

```
#include <boost/signals2.hpp>
#include <iostream>

int main()
{
  boost::signals2::signal<void()> s;
  boost::signals2::connection c = s.connect(
    []{ std::cout << "Hello, world!\n"; });
  s();
  c.disconnect();
}
```

Anstatt für `boost::signals2::signal` `disconnect()` aufzurufen und als Parameter einen Funktionszeiger zu übergeben, kann wie im Beispiel 67.10 geschehen für `boost::signals2::connection` `disconnect()` aufgerufen werden, ohne dass der entsprechende Funktionszeiger übergeben werden muss.
Um eine Funktion kurzzeitig zu blockieren, ohne die Verknüpfung mit dem Signal zu lösen, kann auf `boost::signals2::shared_connection_block` zugegriffen werden.

Beispiel 67.11 Verknüpfungen mit `shared_connection_block` blockieren

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  connection c = s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
  shared_connection_block b{c};
  s();
```

```
  b.unblock();
  s();
}
```

Im Beispiel 67.11 wird die Lambda-Funktion zweimal ausgeführt. Obwohl das Signal **s** dreimal ausgelöst wird, findet beim zweiten Mal kein Aufruf der Lambda-Funktion statt. Denn vor diesem Aufruf wird ein Objekt vom Typ `boost::signals2::shared_connection_block` erstellt, das die Verbindung mit der Lambda-Funktion blockiert. Die Blockierung wird automatisch aufgehoben, wenn der Gültigkeitsbereich endet und das Objekt **b** zerstört wird. Sie können die Blockierung auch explizit aufheben, indem Sie die Methode `unblock()` aufrufen. Da dies im Beispiel 67.11 geschieht, wird beim Auslösen des Signals in der letzten Zeile von `main()` die Lambda-Funktion wieder aufgerufen.
Neben `unblock()` bietet `boost::signals2::shared_connection_block` zwei weitere Methoden `block()` und `blocking()` an. Erstere kann verwendet werden, um eine Verbindung nach einem Aufruf von `unblock()` wieder zu blockieren. Letztere ermöglicht, über den Rückgabewert vom Typ bool abzufragen, ob die Verbindung im Moment blockiert ist.
Beachten Sie, dass die Klasse `boost::signals2::shared_connection_block` shared im Namen trägt. So können mehrere Objekte vom Typ `boost::signals2::shared_connection_block` mit der gleichen Verbindung instanziiert werden.

Beispiel 67.12 Eine Verknüpfung mehrfach blockieren

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  connection c = s.connect([]{ std::cout << "Hello, world!\n"; });
  shared_connection_block b1{c, false};
  {
    shared_connection_block b2{c};
    std::cout << std::boolalpha << b1.blocking() << '\n';
    s();
  }
  s();
}
```

Im Beispiel 67.12 wird zweimal auf **s** zugegriffen. Beim ersten Zugriff findet kein Aufruf der Lambda-Funktion statt, beim zweiten schon. Wenn Sie das Beispiel ausführen, wird `Hello, world!` einmal ausgegeben.
Das erste Objekt vom Typ `boost::signals2::shared_connection_block` blockiert die Verbindung zum Signal **s** nicht, weil dem Konstruktor als zweiter Parameter `false` übergeben wird. Der Aufruf von `blocking()` für das Objekt **b1** gibt entsprechend `false` zurück.
Dennoch wird beim ersten Zugriff auf **s** die Lambda-Funktion nicht ausgeführt. Der Zugriff findet nach der Instanziierung eines zweiten Objekts vom Typ `boost::signals2::shared_connection_block` statt. Da dem Konstruktor kein zweiter Parameter übergeben wird, blockiert dieses Objekt die Verbindung. Die Blockierung wird automatisch aufgehoben, wenn der Gültigkeitsbereich von **b2** endet, so dass beim zweiten Zugriff auf **s** die Lambda-Funktion ausgeführt wird.
Boost.Signals2 bietet die Möglichkeit, eine Verbindung automatisch zu lösen, wenn ein Objekt zerstört wird, dessen Methode mit einem Signal verbunden ist.

Beispiel 67.13 Methoden als Ereignisverarbeiter verwenden

```
#include <boost/signals2.hpp>
#include <memory>
#include <functional>
#include <iostream>

using namespace boost::signals2;

struct world
{
  void hello() const
```

```
  {
    std::cout << "Hello, world!\n";
  }
};

int main()
{
  signal<void()> s;
  {
    std::unique_ptr<world> w(new world());
    s.connect(std::bind(&world::hello, w.get()));
  }
  std::cout << s.num_slots() << '\n';
  s();
}
```

Im Beispiel 67.13 wird eine Methode eines Objekts mit einem Signal verbunden. Dies geschieht mit Hilfe von `std::bind()`. Das Problem ist jedoch, dass das entsprechende Objekt zerstört wird, bevor das Signal ausgelöst wird. Da kein Objekt vom Typ `world` an `std::bind()` übergeben wurde, sondern lediglich dessen Adresse, wird beim Aufruf von `s()` über einen Zeiger auf ein Objekt zugegriffen, das nicht mehr existiert.
Es ist möglich, das Programm so anzupassen, dass die Verbindung automatisch gelöst wird, wenn das Objekt zerstört wird.

Beispiel 67.14 Verknüpfte Methoden automatisch lösen

```
#include <boost/signals2.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::signals2;

struct world
{
  void hello() const
  {
    std::cout << "Hello, world!\n";
  }
};

int main()
{
  signal<void()> s;
  {
    boost::shared_ptr<world> w(new world());
    s.connect(signal<void()>::slot_type(&world::hello, w.get()).track(w));
  }
  std::cout << s.num_slots() << '\n';
  s();
}
```

Wenn Sie Beispiel 67.14 ausführen, stellen Sie fest, dass `num_slots()` `0` zurückgibt. Beim Auslösen des Signals wird nicht versucht, eine Methode für ein Objekt aufzurufen, das bereits zerstört wurde. Die Änderungen, die dazu im Programm notwendig sind: Das Objekt vom Typ `world` muss in einem Smartpointer vom Typ `boost::shared_ptr` verankert werden, der als Parameter an `track()` übergeben wird. Diese Methode wird für den Slot aufgerufen, der an `connect()` übergeben wird, um mitzuteilen, dass das entsprechende Objekt verfolgt werden soll.
Als Slot wird die Funktion oder Methode bezeichnet, die mit einem Signal verknüpft ist. Ein entsprechender Typ, um einen derartigen Slot zu beschreiben, ist in den bisherigen Beispielprogrammen nicht verwendet worden. Das war nicht nötig, weil eine Lambda-Funktion oder ein Zeiger auf eine Funktion oder Methode direkt an `connect()` übergeben werden kann. Der entsprechende Slot wurde im Signal erstellt und musste nicht anderweitig bearbeitet werden.
Im Beispiel 67.14 wird `track()` für den Slot aufgerufen, um den Smartpointer mit diesem zu verbinden. Der Typ des Slots hängt vom Signal ab – die Klasse `boost::signals2::signal` stellt daher einen Typ slot_type

zur Verfügung, um auf den benötigten Typ zugreifen zu können. Da sich slot_type außerdem wie `std::bind()` verhält, können im Beispiel 67.14 die beiden Parameter, die den Slot beschreiben, direkt übergeben werden. Für den entsprechenden Slot, der daraufhin erstellt wird, kann `track()` aufgerufen werden, um den Slot mit einem Smartpointer vom Typ `boost::shared_ptr` zu verknüpfen. Über diesen Smartpointer wird das Objekt verfolgt, was bedeutet: Der Slot wird automatisch entfernt, wenn das im Smartpointer verankerte Objekt zerstört wird.
Slots bieten neben `track()` auch eine Methode `track_foreign()` an, die verwendet werden kann, wenn Objekte in Smartpointern anderer Typen verwaltet werden. Während `track()` einen Smartpointer vom Typ `boost::shared_ptr` erwartet, kann `track_foreign()` verwendet werden, um beispielsweise einen Smartpointer vom Typ `std::shared_ptr` zu übergeben. Andere Smartpointer als `std::shared_ptr` und `std::weak_ptr` müssen Boost.Signals2 explizit bekannt gemacht werden, damit sie an `track_foreign()` übergeben werden können.
Boost.Signals2 erlaubt es, in einem Ereignisverarbeiter auf das Objekt vom Typ `boost::signals2::signal` zuzugreifen, um neue Funktionen zu verknüpfen oder bestehende Verbindungen zu lösen.

Beispiel 67.15 Im Ereignisverarbeiter neue Verknüpfungen erstellen

```
#include <boost/signals2.hpp>
#include <iostream>

boost::signals2::signal<void()> s;

void connect()
{
  s.connect([]{ std::cout << "Hello, world!\n"; });
}

int main()
{
  s.connect(connect);
  s();
}
```

Im Beispiel 67.15 wird innerhalb der Funktion `connect()` auf **`s`** zugegriffen, um eine Lambda-Funktion mit genau diesem Signal zu verknüpfen. Da `connect()` aufgerufen wird, wenn das Signal ausgelöst wird, stellt sich die Frage, ob auch die Lambda-Funktion aufgerufen wird.
Wenn Sie Beispiel 67.15 ausführen, stellen Sie fest, dass keine Ausgabe erfolgt. Die Lambda-Funktion wird nicht aufgerufen. Boost.Signals2 unterstützt das Verknüpfen von Funktionen mit Signalen, während Ereignisverarbeiter ausgeführt werden, die mit diesen Signalen verküpft sind. Neu verknüpfte Funktionen werden aber erst dann ausgeführt, wenn das Signal erneut ausgelöst wird.

Beispiel 67.16 Im Ereignisverarbeiter Verknüpfungen lösen

```
#include <boost/signals2.hpp>
#include <iostream>

boost::signals2::signal<void()> s;

void hello()
{
  std::cout << "Hello, world!\n";
}

void disconnect()
{
  s.disconnect(hello);
}

int main()
{
  s.connect(disconnect);
  s.connect(hello);
  s();
}
```

Im Beispiel 67.16 wird keine neue Funktion im Ereignisverarbeiter verknüpft, sondern eine bestehende Verbindung gelöst. Auch hier stellt sich die Frage, ob `hello()` aufgerufen wird oder nicht.
Auch dieses Beispiel gibt nichts auf die Standardausgabe aus. `hello()` wird nicht aufgerufen.
Sie können sich diese Verhaltensweise erklären, wenn Sie sich vorstellen, dass beim Auslösen eines Signals eine temporäre Kopie aller Slots erstellt wird. Neu mit einem Signal verknüpfte Funktionen werden nicht dieser temporären Kopie hinzugefügt und werden deswegen erst aufgerufen, wenn das Signal neu ausgelöst wird. Verknüpfungen, die aufgelöst wurden, befinden sich zwar in dieser Kopie. Nur wird beim Dereferenzieren der Iteratoren im Combiner überprüft, ob die Verknüpfung gelöst wurde oder nicht. Ein Auflösen der Verknüpfung wird sofort berücksichtigt, was nicht unwichtig ist, wenn das Signal zum Beispiel mit der Methode eines Objekts verknüpft war und dieses zerstört wurde.

67.3 Multithreading

Fast alle Klassen von Boost.Signals2 sind thread-safe. So können Sie zum Beispiel von mehreren Threads aus auf Objekte vom Typ `boost::signals2::signal` und `boost::signals2::connection` zugreifen.
Beachten Sie, dass `boost::signals2::shared_connection_block` nicht thread-safe ist. Das stellt insofern kein Problem dar, als dass Sie mehrere Objekte vom Typ `boost::signals2::shared_connection_block` in unterschiedlichen Threads erstellen können, die alle mit dem gleichen Verbindungsobjekt initialisiert werden können.

Beispiel 67.17 `boost::signals2::signal` ist thread-safe

```
#include <boost/signals2.hpp>
#include <thread>
#include <mutex>
#include <iostream>

boost::signals2::signal<void(int)> s;
std::mutex m;

void loop()
{
  for (int i = 0; i < 100; ++i)
    s(i);
}

int main()
{
  s.connect([](int i){
    std::lock_guard<std::mutex> lock{m};
    std::cout << i << '\n';
  });
  std::thread t1{loop};
  std::thread t2{loop};
  t1.join();
  t2.join();
}
```

Im Beispiel 67.17 wird die Funktion `loop()` in zwei Threads gestartet, in denen jeweils hundertmal auf **`s`** zugegriffen wird, um die mit diesem Objekt verknüpfte Lambda-Funktion aufzurufen. Dieser gleichzeitige Zugriff auf ein Objekt vom Typ `boost::signals2::signal` in mehreren Threads wird von Boost.Signals2 explizit unterstützt.
Beispiel 67.17 gibt Zahlen von 0 bis 99 aus. Da die Variable **`i`** in zwei Threads inkrementiert und per Lambda-Funktion ausgegeben wird, wird nicht nur jede Zahl zweimal ausgegeben. Es kommt außerdem zu Überschneidungen. Entscheidend ist aber, dass es zu keinem Programmabsturz kommen kann, da mehrere Threads auf `boost::signals2::signal` zugreifen dürfen.
Beachten Sie, dass im Beispiel 67.17 dennoch eine Synchronisation stattfindet. Da das Programm zwei Threads verwendet, die beide auf **`s`** zugreifen, wird auch die mit **`s`** verknüpfte Lambda-Funktion in zwei Threads ausgeführt. Damit die Ausgabe innerhalb der Lambda-Funktion jeweils komplett erfolgt und sowohl die Zahl als auch das Zeilenende ausgegeben wird, bevor der andere Thread seine Ausgabe beginnt, wird der Zugriff auf die

Standardausgabe mit einem Mutex synchronisiert. So ist garantiert, dass zu jedem beliebigen Zeitpunkt nur ein Thread **`std::cout`** verwendet und die Ausgabe vom anderen Thread nicht unterbrochen werden kann.
Boost.Signals2 unterstützt standardmäßig Multithreading. Verwenden Sie in einem Programm nur einen einzigen Thread, können Sie diese Unterstützung deaktivieren.

Beispiel 67.18 `boost::signals2::signal` ohne Thread-Safety

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

signal<void()> s;

int main()
{
  typedef keywords::mutex_type<dummy_mutex> dummy_mutex;
  signal_type<void(), dummy_mutex>::type s;
  s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
}
```

`boost::signals2::signal` unterstützt zahlreiche Template-Parameter. Da es der letzte Template-Parameter ist, der den Mutex-Typ definiert, der von `boost::signals2::signal` zur Synchronisierung von Threads verwendet wird, bietet Boost.Signals2 eine einfachere Möglichkeit an, als die vollständige Liste aller Template-Parameter an `boost::signals2::signal` übergeben zu müssen.
Im Namensraum `boost::signals2::keywords` sind Klassen definiert, mit denen Template-Parameter mit Namen angegeben werden können. So kann auf `boost::signals2::keywords::mutex_type` zugegriffen werden, um den Mutex-Typ als zweiten Template-Parameter an `boost::signals2::signal_type` zu übergeben. Beachten Sie, dass in diesem Fall auf `boost::signals2::signal_type` und nicht auf `boost::signals2::signal` zugegriffen werden muss. Über `boost::signals2::signal_type::type` wird der Typ erhalten, der `boost::signals2::signal` entspricht und notwendig ist, um das Signal **`s`** zu definieren.
Boost.Signals2 stellt mit `boost::signals2::dummy_mutex` eine Mutex-Implementation zur Verfügung, die leer ist und nichts macht. Wird wie im Beispiel 67.18 ein Signal mit dieser Klasse definiert, unterstützt es kein Multithreading.

Kapitel 68

Boost.MetaStateMachine

Boost.MetaStateMachine wird verwendet, um Zustandsautomaten zu definieren. Zustandsautomaten beschreiben Objekte über ihre Zustände. So legen sie fest, welche Zustände es gibt und von welchem Zustand in welchen anderen gewechselt werden kann.
Boost.MetaStateMachine bietet drei verschiedene Möglichkeiten an, Zustandsautomaten zu definieren. In der Dokumentation von Boost.MetaStateMachine ist von Frontends die Rede. Vom Frontend hängt es ab, welchen Code Sie schreiben müssen, um einen Zustandsautomaten zu erstellen.
Wenn Sie sich für das einfache Frontend oder das Funktor-Frontend entscheiden, definieren Sie Zustandsautomaten auf eine eher herkömmliche Art und Weise: Sie erstellen Klassen, leiten diese von Klassen ab, die Boost.MetaStateMachine zur Verfügung stellt, definieren erforderliche Eigenschaften und schreiben so den notwendigen C++-Code selbst.
Der grundlegende Unterschied zwischem dem einfachen Frontend und dem Funktor-Frondend ist, dass Sie dort, wo Funktionen erwartet werden, im Funktor-Frontend Funktionsobjekte angeben können, während Sie im einfachen Frontend Funktionszeiger verwenden müssen.
Das dritte Frontend heißt eUML und basiert auf einer domainspezifischen Sprache. So können Sie mit diesem Frontend Zustandsautomaten definieren, indem Sie Definitionen aus Zustandsautomaten der UML entnehmen. Entwickler, die mit der UML vertraut sind, können Definitionen aus entsprechenden UML-Verhaltensdiagrammen in C++-Code übernehmen. Eine Übersetzung von UML-Definitionen in herkömmlichen C++-Code entfällt.
eUML basiert auf zahlreichen Makros, die Sie verwenden müssen, wenn Sie mit diesem Frontend arbeiten. Die Makros haben den Vorteil, dass Sie mit vielen Klassen, die Boost.MetaStateMachine verwendet und in Ihrem Code erwartet, nicht in Berührung kommen. Sie müssen zwar die entsprechenden Makros kennen. Dafür kann es nicht passieren, dass Sie wie beim einfachen Frontend oder dem Funktor-Frontend vergessen, Ihren Zustandsautomaten von einer bestimmten Klasse aus Boost.MetaStateMachine abzuleiten. Im Folgenden lernen Sie Boost.MetaStateMac mit dem Frontend eUML kennen.

Beispiel 68.1 Ein einfacher Zustandsautomat mit eUML

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)
```

```
int main()
{
  msm::back::state_machine<light_state_machine> light;
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
  std::cout << *light.current_state() << '\n';
}
```

Beispiel 68.1 verwendet den einfachsten Zustandsautomaten, den es gibt: Eine Lampe kennt genau zwei Zustände. Sie ist entweder an oder aus. Ist sie aus, kann sie eingeschaltet werden. Ist sie an, kann sie ausgeschaltet werden. Es kann also von jedem Zustand in den anderen gewechselt werden.
Im Beispiel 68.1 wird das Frontend eUML verwendet, um den Zustandsautomaten einer Lampe zu beschreiben. Da es für Boost.MetaStateMachine keine Master-Headerdatei gibt, die alle notwendigen Definitionen zur Verfügung stellt, müssen Sie jeweils benötigte Headerdateien einzeln einbinden. So erhalten Sie über `boost/msm/front/euml/euml.hpp` und `boost/msm/front/euml/state_grammar.hpp` Zugriff auf Makros der eUML. `boost/msm/back/state_machine.hpp` müssen Sie einbinden, um einen Zustandsautomaten aus dem Frontend mit einem Zustandsautomaten aus dem Backend verbinden zu können. Während Frontends verschiedene Möglichkeiten zur Verfügung stellen, Zustandsautomaten zu beschreiben, ist im Backend die eigentliche Funktionalität enthalten, die Zustandsautomaten ausmacht. Es existiert lediglich ein Backend in Boost.MetaStateMachine, so dass Sie sich nicht wie bei Frontends für eine bestimmte Implementation entscheiden müssen.
Alle Definitionen von Boost.MetaStateMachine befinden sich im Namensraum `boost::msm`. Leider beziehen sich viele Makros der eUML nicht explizit auf Klassen in diesem Namensraum. Entweder wird auf den Namensraum `msn` Bezug genommen oder es werden Klassen ohne Angabe eines Namensraums verwendet. Sie müssen daher wie im Beispiel 68.1 geschehen die Namensräume `boost::msn` und `boost::msm::front::euml` besonders behandeln. Die Makros der eUML führen ansonsten zu Compilerfehlern.
Um den Zustandsautomaten einer Lampe zu verwenden, werden zuerst zwei Zustände für an und aus definiert. Sie verwenden dazu das Makro `BOOST_MSM_EUML_STATE`, dem Sie den Namen des Zustands als zweiten Parameter übergeben. Über den ersten Parameter können Sie den Zustand beschreiben. Sie erfahren später, welche Angaben Sie hier vornehmen können. Die beiden im Beispiel 68.1 definierten Zustände heißen **`Off`** und **`On`**.
Um zwischen Zuständen wechseln zu können, benötigen Sie Ereignisse. Sie verwenden dazu das Makro `BOOST_MSM_EUML_EVENT`, dem Sie ausschließlich den Namen des Ereignisses übergeben. Im Beispiel 68.1 wird ein Ereignis **`press`** definiert. Es symbolisiert das Drücken eines Lichtschalters. Da über das gleiche Ereignis ein Licht an- und ausgemacht werden kann, wird im Code lediglich ein Ereignis definiert.
Haben Sie die benötigten Zustände und Ereignisse definiert, greifen Sie auf das Makro `BOOST_MSM_EUML_TRANSITION_TABLE` zu, um eine *Zustandsübergangstabelle* zu erstellen. Diese beschreibt, von welchem Zustand in welchen anderen gewechselt wird, wenn ein bestimmtes Ereignis eintritt.
`BOOST_MSM_EUML_TRANSITION_TABLE` erwartet zwei Parameter, von denen der zweite der Name der Zustandsübergangstabelle ist. Der erste Parameter definiert die Zustandsübergangstabelle. Die Syntax des ersten Parameters ist so gestaltet, dass Sie auf einen Blick erkennen sollten, wie Zustände und Ereignisse zusammenhängen. So bedeutet im Beispiel 68.1 `Off + press ==On`, dass der Automat im Zustand **`Off`** bei Eintreten des Ereignisses **`press`** in den Zustand **`On`** wechselt. Das Frontend eUML spielt an dieser Stelle seine Stärken aus, da eine eingängige und selbsterklärende Syntax zur Definition der Zustandsübergangstabelle verwendet werden kann.
Nachdem Sie die Zustandsübergangstabelle definiert haben, definieren Sie den Zustandsautomaten. Sie verwenden dazu das Makro `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`. Der zweite Parameter ist wieder der einfachere: Er legt den Namen des Zustandsautomaten fest. Der Zustandsautomat im Beispiel 68.1 heißt `light_state_machine`.
Der erste Parameter von `BOOST_MSM_EUML_DECLARE_STATE_MACHINE` ist ein Tuple. Als erster Wert wird der Name der Zustandsübergangstabelle angegeben. Der zweite Wert ist ein Ausdruck, in dem auf ein Attribut **`init_`** zugegriffen wird, das von Boost.MetaStateMachine zur Verfügung gestellt wird. Was es mit Attributen auf sich hat, erfahren Sie später. Der Ausdruck `init_ << Off` ist notwendig, um den Anfangszustand des Automaten auf **`Off`** zu setzen.
Bei dem mit `BOOST_MSM_EUML_DECLARE_STATE_MACHINE` definierten Zustandsautomaten `light_state_machine` handelt es sich um eine Klasse. Sie müssen die Klasse verwenden, um einen Zustandsautomaten aus dem Backend zu instanziieren. Im Beispiel 68.1 wird dazu auf die Template-Klasse `boost::msm::back::state_machine` zugegriffen und ihr als Parameter `light_state_machine` übergeben. Sie erhalten nun eine Instanz des

Zustandsautomaten namens **`light`**.
Zustandsautomaten bieten mit `process_event()` eine Methode an, die Ereignisse verarbeiten kann. Sie übergeben `process_event()` als einzigen Parameter ein Ereignis, woraufhin der Automat abhängig von seiner Zustandsübergangstabelle den Zustand wechselt.
Damit Sie nachverfolgen können, was im Beispiel 68.1 passiert, wenn mehrfach `process_event()` aufgerufen wird, wird außerdem auf `current_state()` zugegriffen. Diese Methode sollten Sie ausschließlich zu Debuggingzwecken verwenden. Sie gibt einen Zeiger auf ein int zurück. Jedem Zustand ist eine Zahl zugewiesen, und zwar in der Reihenfolge, in der auf Zustände innerhalb von `BOOST_MSM_EUML_TRANSITION_TABLE` zugegriffen wird. Im Beispiel 68.1 ist entsprechend **`Off`** 0 und **`On`** 1 zugewiesen. Wenn Sie das Beispiel ausführen, wird `0`, `1` und `0` auf die Standardausgabe ausgegeben. Zuerst ist das Licht aus. Dann wird der Lichtschalter zweimal betätigt, woraufhin das Licht an und wieder aus geht.

Beispiel 68.2 Ein Zustandsautomat mit Status, Ereignis und Aktion

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    BOOST_MSM_EUML_STATE_NAME(Off) &sourceState,
    BOOST_MSM_EUML_STATE_NAME(On) &targetState) const
  {
    std::cout << "Switching on\n";
  }

  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    decltype(On) &sourceState,
    decltype(Off) &targetState) const
  {
    std::cout << "Switching off\n";
  }
};

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Beispiel 68.2 erweitert den Zustandsautomaten für die Lampe um eine Aktion. Eine Aktion wird ausgeführt, wenn ein Ereignis eintritt und ein Zustandswechsel erfolgt. Da Aktionen optional sind, konnte der Zustandsauto-

mat im vorherigen Beispiel ohne sie auskommen.
Sie definieren eine Aktion mit `BOOST_MSM_EUML_ACTION`. Genaugenommen definieren Sie ein Funktionsobjekt. Sie müssen daher den Operator `operator()` überladen. Der Operator muss vier Parameter akzeptieren, bei denen es sich um Referenzen auf ein Ereignis, einen Zustandsautomaten und zwei Zustände handeln muss. Ob Sie ein Template definieren oder für alle vier Parameter konkrete Typen angeben, bleibt Ihnen überlassen. Im Beispiel 68.2 sind lediglich für die letzten beiden Parameter konkrete Typen angegeben. Da diese Parameter beschreiben, von welchem Zustand in welchen anderen gewechselt wird, können Sie `operator()` so überladen, dass unterschiedliche Methoden für unterschiedliche Wechsel aufgerufen werden.
Beachten Sie, dass es sich bei den Zuständen **`On`** und **`Off`** um Objekte handelt. Boost.MetaStateMachine bietet mit `BOOST_MSM_EUML_STATE_NAME` ein Makro an, um den Typ von Zustandsobjekten zu erhalten. Arbeiten Sie mit C++11, können Sie anstelle des Makros den Operator `decltype` verwenden.
Die mit `BOOST_MSM_EUML_ACTION` definierte Aktion **`switch_light`** soll ausgeführt werden, wenn der Lichtschalter betätigt wird. Dazu wurde die Zustandsübergangstabelle geändert. Die erste Angabe lautet nun `Off + press /switch_light ==On`. Sie geben Aktionen hinter einem Schrägstrich nach dem Ereignis an. Diese Angabe bedeutet, dass der Operator `operator()` von **`switch_light`** aufgerufen wird, wenn sich der Automat im Zustand **`Off`** befindet und das Ereignis **`press`** eintritt. Nachdem die Aktion ausgeführt wurde, befindet sich der Automat im Zustand **`On`**.
Wenn Sie Beispiel 68.2 ausführen, wird zuerst `Switching on` und dann `Switching off` auf die Standardausgabe ausgegeben.

Beispiel 68.3 Ein Zustandsautomat mit Status, Ereignis, Guard und Aktion

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return true;
  }
};

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm, class Source, class Target>
  void operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    std::cout << "Switching\n";
  }
};

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
```

```
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Im Beispiel 68.3 wird in der Zustandsübergangstabelle ein Guard verwendet. Die erste Zeile lautet `Off + press [!is_broken] /switch_light ==On`. Die Angabe von **`is_broken`** in eckigen Klammern bewirkt, dass vor dem Aufruf der Aktion **`switch_light`** überprüft wird, ob der Zustandswechsel erfolgen darf. Dies wird als Guard bezeichnet. Dieser muss ein Ergebnis vom Typ bool zurückgeben.
Ein Guard wie **`is_broken`** wird wie Aktionen mit dem Makro `BOOST_MSM_EUML_ACTION` definiert. Es muss demnach auch der Operator `operator()` überladen werden, der die vier bekannten Parameter erwartet. Wichtig ist, dass `operator()` einen Rückgabewert vom Typ bool hat. Ist dies der Fall, kann die Aktion als Guard verwendet werden.
Beachten Sie, dass Sie logische Operatoren wie `operator!` verwenden dürfen, wenn Sie einen Guard in eckigen Klammern angeben.
Wenn Sie das Beispiel ausführen, sehen Sie, dass keine Ausgabe auf die Standardausgabe erfolgt. Die Aktion **`switch_light`** wird nicht ausgeführt – das Licht bleibt aus. Der Guard **`is_broken`** gibt zwar `true` zurück. Aus dem `true` wird jedoch aufgrund des Operators `operator!` ein `false`.
Sie können Guards verwenden, um zu überprüfen, ob ein Zustandswechsel erfolgen darf. Im Beispiel 68.3 soll mit **`is_broken`** angegeben werden, dass die Lampe kaputt ist. Auch wenn grundsätzlich ein Wechsel von aus zu an möglich ist und der Zustandsautomat Lampen richtig beschreibt, kann die im Beispiel verwendete Lampe nicht eingeschaltet werden. Der Zustand der Lampe **`light`** ist trotz zweimaligem Aufruf von `process_event()` **`Off`**.

Beispiel 68.4 Ein Zustandsautomat mit Status, Ereignis und Ein- und Austrittsaktion

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Entering\n";
  }
};

BOOST_MSM_EUML_ACTION(state_exit)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Exiting\n";
  }
};

BOOST_MSM_EUML_STATE((state_entry, state_exit), Off)
BOOST_MSM_EUML_STATE((state_entry, state_exit), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
```

```
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Im Beispiel 68.4 wird als erster Parameter an `BOOST_MSM_EUML_STATE` ein Tuple bestehend aus `state_entry` und `state_exit` übergeben. Es handelt sich hierbei um Ein- und Austrittsaktionen. Die erste Angabe ist die Eintrittsaktion, die zweite die Austrittsaktion. Die Aktionen werden ausgeführt, wenn in einen Zustand gewechselt wird oder wenn ein Zustand verlassen wird.
Ein- und Austrittsaktionen werden wie Aktionen mit dem Makro `BOOST_MSM_EUML_ACTION` definiert. Der überladene Operator `operator()` erwartet jedoch mit einer Referenz auf ein Ereignis, einen Zustandsautomaten und einen Zustand lediglich drei Parameter. Da bei Ein- und Austrittsaktionen nicht der Übergang von einem Zustand zu einem anderen zählt, wird an den überladenen Operator `operator()` lediglich ein Zustand übergeben. Für Eintrittsaktionen ist das der Zustand, zu dem gewechselt wird, für Austrittsaktionen der Zustand, der verlassen wird.
Im Beispiel 68.4 besitzen beide Zustände **`Off`** und **`On`** Ein- und Austrittsaktionen. Da das Ereignis **`press`** zweimal eintritt, wird zweimal `Entering` und `Exiting` ausgegeben. Beachten Sie, dass zuerst `Exiting` und dann `Entering` ausgegeben wird. Die erste Aktion, die ausgeführt wird, ist eine Austrittsaktion.
Beim ersten Ereignis **`press`** wird vom Ursprungszustand **`Off`** zu **`On`** gewechselt. Für diesen Zustandswechsel wird einmal `Exiting` und einmal `Entering` ausgegeben. Beim zweiten Ereignis **`press`** wird von **`On`** zu **`Off`** gewechselt. Es wird wiederum einmal `Exiting` und einmal `Entering` ausgegeben. Bei einem Zustandswechel wird zuerst die Austrittsaktion des alten Zustands und dann die Eintrittsaktion des neuen Zustands ausgeführt.

Beispiel 68.5 Attribute in Zustandsautomaten

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Switched on\n";
    ++fsm.get_attribute(switched_on);
  }
};
BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return fsm.get_attribute(switched_on) > 1;
  }
};

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((state_entry), On)
BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] == On,
  On + press == Off
```

```
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Im Beispiel 68.5 wird der bereits bekannte Guard **is_broken** eingesetzt, um zu überprüfen, ob ein Zustandswechsel von **Off** zu **On** möglich ist. Diesmal hängt der Rückgabewert von **is_broken** davon ab, wie oft der Lichtschalter betätigt wurde. So soll die Lampe zweimal eingeschaltet werden können, bevor sie kaputt geht. Um mitzuzählen, wie oft die Lampe eingeschaltet wurde, wird ein Attribut verwendet.
Attribute sind Variablen, die Objekten hinzugefügt werden können. Sie sind wichtig, um das Verhalten von Zustandsautomaten zur Laufzeit zu verändern. Da Daten wie beispielsweise die Anzahl der Einschaltvorgänge irgendwo gespeichert werden können müssen, bietet es sich an, sie direkt im Zustandsautomaten, einem Zustand oder einem Ereignis zu speichern.
Bevor ein Attribut verwendet werden kann, muss es definiert werden. Dies geschieht mit dem Makro `BOOST_MSM_EUML_DECLARE_ATTRIBUTE`. Der erste Parameter, der an `BOOST_MSM_EUML_DECLARE_ATTRIBUTE` übergeben wird, ist der Typ, der zweite Parameter der Name des Attributs. Im Beispiel 68.5 wird auf diese Weise ein Attribut **switched_on** vom Typ int definiert.
Nachdem ein Attribut definiert wurde, muss es einem Objekt hinzugefügt werden. Im Beispiel wird das Attribut **switched_on** dem Zustandsautomaten hinzugefügt. Dies geschieht über den fünften Wert im Tuple, der als erster Parameter an `BOOST_MSM_EUML_DECLARE_STATE_MACHINE` übergeben wird. Dabei wird mit **attributes_** auf ein von Boost.MetaStateMachine zur Verfügung gestelltes Schlüsselwort zugegriffen, um eine Lambda-Funktion zu erstellen. Indem **attributes_** wie ein Stream verwendet wird und **switched_on** über den Operator `operator<<` an **attributes_** übergeben wird, wird **switched_on** als Attribut dem Zustandsautomaten hinzugefügt.
Beachten Sie, dass **no_action** als dritter und vierter Wert im Tuple angegeben ist. Dies ist notwendig, um als fünfte Angabe im Tuple das Attribut hinzufügen zu können. Der dritte und vierte Wert können verwendet werden, um Ein- und Austrittsaktionen für einen Zustandsautomaten zu definieren. Sollen keine Ein- und Austrittsaktionen verwendet werden, muss auf **no_action** zugegriffen werden.
Nachdem das Attribut dem Zustandsautomaten hinzugefügt wurde, kann mit `get_attribute()` auf das Attribut zugegriffen werden. So wird diese Methode im Beispiel 68.5 in der Eintrittsaktion `state_entry` aufgerufen, um den Wert des Attributs um eins zu erhöhen. Da `state_entry` nur mit dem Zustand **On** verbunden ist, wird **switched_on** immer dann um eins erhöht, wenn das Licht eingeschaltet wird.
Neben `state_entry` wird außerdem im Guard **is_broken** auf **switched_on** zugegriffen. **is_broken** überprüfen, ob der Wert des Attributs größer als 1 ist. Ist dies der Fall, gibt der Guard `true` zurück. Da Attribute mit dem Standardkonstruktor initialisiert werden und **switched_on** auf 0 gesetzt ist, gibt **is_broken** dann `true` zurück, wenn das Licht zweimal eingeschaltet wurde.
Wenn Sie Beispiel 68.5 ausführen, tritt fünfmal das Ereignis **press** ein. Das Licht wird zweimal ein- und ausgeschaltet und anschließend wieder eingeschaltet. Für die ersten beiden Einschaltvorgänge wird `Switched on` ausgegeben. Beim dritten Einschalten findet jedoch keine Ausgabe statt. Da die Ausgabe in der Eintrittsaktion des Zustands **On** erfolgt, nach einem zweimaligen Einschalten **is_broken** jedoch `true` zurückgibt, findet kein Zustandswechsel mehr von **Off** zu **On** statt. Dementsprechend wird beim dritten Einschalten die Eintrittsaktion nicht ausgeführt.

Beispiel 68.6 Zugriff auf Attribute in Zustandsübergangstabellen

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
```

```
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

void write_message()
{
  std::cout << "Switched on\n";
}

BOOST_MSM_EUML_FUNCTION(WriteMessage_, write_message, write_message_,
  void, void)

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [fsm_(switched_on) < Int_<2>()] / (++fsm_(switched_on),
    write_message_()) == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Beispiel 68.6 macht das Gleiche wie das vorherige: Nach zweimaligem Einschalten ist die Lampe kaputt und kann kein weiteres Mal eingeschaltet werden. Während im vorherigen Beispiel auf das Attribut **`switched_on`** in Aktionen zugegriffen wurde, findet der Zugriff in diesem Beispiel direkt in der Zustandsübergangstabelle statt. Boost.MetaStateMachine bietet die Funktion `fsm_()` an, um auf ein Attribut in einem Zustandsautomaten zuzugreifen. So wird auf diese Weise ein Guard definiert, der überprüft, ob **`switched_on`** kleiner als 2 ist. Außerdem wird eine Aktion definiert, die **`switched_on`** jedes Mal um eins erhöht, wenn ein Zustandswechsel von **`Off`** zu **`On`** stattfindet.
Beachten Sie, dass der Kleiner-als-Vergleich im Guard mit `Int_<2>()` stattfindet. Die Zahl 2 muss als Template-Parameter an `Int_` übergeben werden, um dann eine Instanz dieser Klasse zu erstellen. Auf diese Weise wird ein Funktionsobjekt erstellt, das den richtigen Typ hat, damit Boost.MetaStateMachine erkennt, dass ein Kleiner-als-Vergleich mit 2 durchgeführt werden soll.
Im Beispiel 68.6 kommt mit `BOOST_MSM_EUML_FUNCTION` außerdem ein Makro zum Einsatz, um eine Funktion als Aktion zu definieren. Dem Makro wird als erster Parameter der Name der Aktion übergeben, mit dem sie im Funktion-Frontend eingesetzt werden kann. Der zweite Parameter ist der Name der Funktion. Der dritte Parameter ist der Name der Aktion, mit dem sie in der eUML eingesetzt wird. Der vierte und fünfte Parameter sind der Rückgabewert der Funktion – einmal für den Fall, dass die Aktion für einen Zustandsübergang eingesetzt wird, einmal für den Fall, dass die Aktion einen Ein- oder Austritt beschreibt. Nachdem `write_message()` auf diese Weise in eine Aktion umgewandelt wurde, wird ein Objekt vom Typ `write_message_` erstellt und dieses hinter `++fsm_(switched_on)` in der Zustandsübergangstabelle angegeben. Bei einem Übergang von **`Off`** zu **`On`** wird zuerst das Attribut **`switched_on`** inkrementiert und dann `write_message()` aufgerufen.
Wenn Sie Beispiel 68.6 ausführen, wird wie im vorherigen Beispiel `Switched on` genau zweimal ausgegeben. Boost.MetaStateMachine bietet neben `fsm_()` weitere Funktionen wie `state_()` oder `event_()` an, um auf Attribute zuzugreifen, die anderen Objekten hinzugefügt wurden. Mit `Char_` und `String_` stehen weitere Klassen zur Verfügung, die ähnlich wie `Int_` eingesetzt werden können.

Tipp

Das Frontend eUML verlangt wie in den Beispielen zu sehen den Einsatz vieler Makros. Die Headerdatei `boost/msm/front/euml/common.hpp` gibt einen guten Überblick über diese – dort finden Sie die Definition aller eUML-Makros.

Teil XVII

Sonstige Bibliotheken

Die folgenden Bibliotheken stellen kleine, aber nützliche Hilfsmittel zur Verfügung.

- Boost.Utility ist die sonstige Bibliothek in den Boost-Bibliotheken: Alles, was nicht zu anderen Bibliotheken passt, wird in Boost.Utility abgelegt.
- Boost.Assign bietet Hilfsfunktionen an, um zum Beispiel mehrere Werte einfacher als durch einen wiederholten Aufruf von `push_back()` einem Container hinzuzufügen.
- Boost.Swap bietet eine für die Boost-Bibliotheken optimierte Variante von `std::swap()` an.
- Boost.Operators macht es einfach, Operatoren basierend auf anderen zu überladen.

Kapitel 69

Boost.Utility

Die Bibliothek Boost.Utility ist ein Sammelsurium verschiedener Klassen und Funktionen, die sich als sehr nützlich herausgestellt haben, jedoch zu klein sind, um in eigenen Bibliotheken gepflegt zu werden. Da die von Boost.Utility angebotenen Hilfsmittel so klein sind, lassen sie sich schnell erlernen. Weil Boost.Utility jedoch ein Sammelbecken für alles Mögliche ist, hängen die Hilfsmittel in keiner Weise zusammen.
Wie in diesem Buch üblich soll anhand von Beispielen vorgestellt werden, was die Bibliothek zu bieten hat. Weil sich in Boost.Utility verschiedenste und voneinander unabhängige Hilfsmittel befinden, bauen die folgenden Beispiele jedoch nicht aufeinander auf.
Boost.Utility stellt mit `boost/utility.hpp` eine Headerdatei zur Verfügung, über die auf verschiedene Hilfsmittel zugegriffen werden kann. Da jedoch nicht alle Hilfsmittel über `boost/utility.hpp` eingebunden werden, wird in den folgenden Beispielprogrammen jeweils auf genau die Headerdatei zugegriffen, in der das jeweilige Hilfsmittel definiert ist.

> **Anmerkung**
>
> Mit Boost 1.55.0 wurde eine neue Bibliothek namens Boost.Core eingeführt, in die einige Hilfsmittel aus Boost.Utility übertragen wurden. So befindet sich zum Beispiel `boost::checked_delete()` seit dieser Version in Boost.Core. Da es sich lediglich um eine Reorganisation in den Boost-Bibliotheken handelt, haben sich weder die Hilfsmittel an sich noch die Headerdateien, in denen sie sich befinden, geändert. So können Sie die Beispiele in diesem Kapitel auch mit Boost 1.55.0 oder einer neueren Version kompilieren.

Beispiel 69.1 `boost::checked_delete()` in Aktion

```
#include <boost/checked_delete.hpp>
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : public boost::intrusive::list_base_hook<>
{
  std::string name_;
  int legs_;

  animal(std::string name, int legs) : name_{std::move(name)},
    legs_{legs} {}
};

int main()
{
  animal *a = new animal{"cat", 4};

  typedef boost::intrusive::list<animal> animal_list;
```

```
  animal_list al;

  al.push_back(*a);

  al.pop_back_and_dispose(boost::checked_delete<animal>);
  std::cout << al.size() << '\n';
}
```

Im Beispiel 69.1 wird eine Funktion `boost::checked_delete()` als Parameter an die Methode `pop_back_and_dispose()` übergeben, die die in der Bibliothek Boost.Intrusive definierte Klasse `boost::intrusive::list` zur Verfügung stellt. Während `boost::intrusive::list` und `pop_back_and_dispose()` im Kapitel 18 vorgestellt werden, handelt es sich bei `boost::checked_delete()` um eine von Boost.Utility zur Verfügung gestellte Funktion. Um diese Funktion nutzen zu können, muss die Headerdatei `boost/checked_delete.hpp` eingebunden werden.
`boost::checked_delete()` erwartet als einzigen Parameter einen Zeiger auf ein Objekt, das mit `delete` zerstört wird. Da an `pop_back_and_dispose()` eine Funktion übergeben werden muss, die einen Zeiger als einzigen Parameter erwartet, um das entsprechende Objekt zu zerstören, bietet es sich an, auf die von Boost.Utility zur Verfügung gestellte Funktion zuzugreifen. So muss die Funktion nicht selbst definiert werden.
Der Unterschied zwischen `boost::checked_delete()` und einer selbst definierten Funktion, die lediglich auf `delete` zugreift, ist: `boost::checked_delete()` stellt sicher, dass der Typ des Objekts, das zerstört wird, vollständig ist. `delete` hingegen akzeptiert auch Zeiger auf Objekte, deren Typ nicht vollständig ist. Es handelt sich hierbei um ein Detail aus dem C++-Standard, das Sie ignorieren können, wenn es Ihnen nicht bekannt ist. Der Vollständigkeit halber muss jedoch erwähnt werden, dass `boost::checked_delete()` nicht völlig identisch ist mit einem Aufruf von `delete`, sondern etwas höhere Anforderungen stellt.
Neben `boost::checked_delete()` stellt Boost.Utility `boost::checked_array_delete()` zur Verfügung. Diese Funktion muss verwendet werden, wenn ein Array zerstört werden soll. Sie ruft `delete[]` statt `delete` auf. Darüber hinaus stehen mit `boost::checked_deleter` und `boost::checked_array_deleter` zwei Klassen zur Verfügung, um Funktionsobjekte zu erstellen, die sich so verhalten wie `boost::checked_delete()` und `boost::checked_array_delete()`.

Beispiel 69.2 `BOOST_CURRENT_FUNCTION` in Aktion

```
#include <boost/current_function.hpp>
#include <iostream>

int main()
{
  const char *funcname = BOOST_CURRENT_FUNCTION;
  std::cout << funcname << '\n';
}
```

Im Beispiel 69.2 wird ein Makro namens `BOOST_CURRENT_FUNCTION` verwendet, mit dem der Name einer Funktion als Zeichenkette erhalten werden kann – und zwar der Funktion, in der sich das Makro befindet. Dazu muss die Headerdatei `boost/current_function.hpp` eingebunden werden.
`BOOST_CURRENT_FUNCTION` stellt eine plattformunabhängige Möglichkeit dar, den Namen einer Funktion zu erhalten. Seit C++11 können Sie auf das standardisierte Makro `__func__` zugreifen. Vor C++11 boten Compiler wie Visual C++ oder GCC das Makro `__FUNCTION__` an. Da es sich hierbei um eine Erweiterung dieser Compiler handelte und nicht um ein Makro, das im Standard definiert war, half `BOOST_CURRENT_FUNCTION`, plattformunabhängigen Code zu schreiben.
Beispiel 69.2 unter Windows ausgeführt und mit Visual C++ 2013 übersetzt gibt `int __cdecl main(void)` aus.

Beispiel 69.3 `boost::prior()` und `boost::next()` in Aktion

```
#include <boost/next_prior.hpp>
#include <array>
#include <algorithm>
#include <iostream>

int main()
{
  std::array<char, 4> a{{'a', 'c', 'b', 'd'}};

  auto it = std::find(a.begin(), a.end(), 'b');
```

```
  auto prior = boost::prior(it, 2);
  auto next = boost::next(it);

  std::cout << *prior << '\n';
  std::cout << *it << '\n';
  std::cout << *next << '\n';
}
```

Mit `boost::prior()` und `boost::next()` stellt Boost.Utility zwei einfache Funktionen zur Verfügung, um einen Iterator relativ zu einem anderen zu erhalten. Während **`it`** im Beispiel 69.3 auf den Buchstaben „b“ im Array zeigt, wird mit **`prior`** auf „a“ und mit **`next`** auf „d“ verwiesen.
Der Unterschied zwischen `boost::prior()` und `boost::next()` auf der einen und `std::advance()` auf der anderen Seite ist, dass die von Boost.Utility angebotenen Funktionen den Iterator, der als Parameter übergeben wird, nicht verändern. Stattdessen geben Sie einen neuen Iterator zurück.
Beide Funktionen `boost::prior()` und `boost::next()` akzeptieren neben einem Iterator einen zweiten Parameter. Dieser Parameter gibt an, wie viele Schritte der Iterator vor- oder zurückspringen soll. Während `boost::prior()` im Beispiel 69.3 den Iterator zwei Schritte nach hinten setzt, führt der Aufruf von `boost::next()` dazu, dass der Iterator einen Schritt nach vorne macht.
Beachten Sie, dass Sie bei Angabe der Schritte in jedem Fall eine positive Zahl angeben müssen – auch dann, wenn Sie mit `boost::prior()` Schritte nach hinten machen.
Um `boost::prior()` und `boost::next()` nutzen zu können, müssen Sie die Headerdatei `boost/next_prior.hpp` einbinden.
Die beiden Funktionen sind mit C++11 in die Standardbibliothek aufgenommen worden. Sie heißen dort `std::prev()` und `std::next()` und sind in der Headerdatei `iterator` definiert.

Beispiel 69.4 `boost::noncopyable` in Aktion

```
#include <boost/noncopyable.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : boost::noncopyable
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

void print(const animal &a)
{
  std::cout << a.name << '\n';
  std::cout << a.legs << '\n';
}

int main()
{
  animal a{"cat", 4};
  print(a);
}
```

Boost.Utility bietet eine Klasse `boost::noncopyable` an, die verhindert, dass Objekte kopiert werden können. Dazu muss lediglich von `boost::noncopyable` abgeleitet werden. Diese Klasse ist in der Headerdatei `boost/noncopyable.hpp` definiert.
Der Effekt ist der Gleiche, wie wenn der Copy-Konstruktor und der Zuweisungsoperator als private Methoden definiert werden oder – was seit C++11 möglich ist – mit `delete` explizit aus einer Klasse entfernt werden. Wird `boost::noncopyable` als Elternklasse verwendet, kann jedoch unter Umständen die Lesbarkeit des Codes verbessert werden, da deutlicher erkennbar ist, dass Objekte einer Klasse nicht kopiert werden dürfen.

> **Anmerkung**
>
> Einige Entwickler ziehen `boost::noncopyable` vor, andere explizit mit `delete` entfernte Methoden. Sie finden zum Beispiel auf Stack Overflow Argumente beider Seiten.

Beispiel 69.4 kann kompiliert und ausgeführt werden. Würde jedoch der Funktionskopf von `print()` geändert werden, so dass der Parameter keine Referenz mehr wäre, sondern ein Objekt vom Typ `animal` als Kopie erwartet werden würde, würde der Compiler einen Fehler melden.

Beispiel 69.5 `boost::addressof()` in Aktion

```
#include <boost/utility/addressof.hpp>
#include <string>
#include <iostream>

struct animal
{
  std::string name;
  int legs;

  int operator&() const { return legs; }
};

int main()
{
  animal a{"cat", 4};
  std::cout << &a << '\n';
  std::cout << boost::addressof(a) << '\n';
}
```

Mit `boost::addressof()` stellt Boost.Utility eine Funktion zur Verfügung, um die Adresse eines Objekts zu erhalten – auch dann, wenn wie im Beispiel 69.5 der Operator `operator&` überladen wurde.
Um `boost::addressof()` einsetzen zu können, müssen Sie die Headerdatei `boost/utility/addressof.hpp` einbinden.
Die Funktion ist mit C++11 in die Standardbibliothek aufgenommen wurden und steht als `std::addressof()` in der Headerdatei `memory` zur Verfügung.

Beispiel 69.6 `BOOST_BINARY` in Aktion

```
#include <boost/utility/binary.hpp>
#include <iostream>

int main()
{
  int i = BOOST_BINARY(1001 0001);
  std::cout << i << '\n';

  short s = BOOST_BINARY(1000 0000 0000 0000);
  std::cout << s << '\n';
}
```

Boost.Utility bietet mit `BOOST_BINARY` ein Makro an, um Zahlen im Binärformat anzugeben. Denn während mit Präfixen wie `0x` und `0` Zahlen im Hexadezimal- und Oktalformat angegeben werden konnten, unterstützte C++ lange kein Binärformat. Das hat sich erst mit C++14 geändert, das das Präfix `0b` kennt.
Beispiel 69.6 gibt `145` und `-32768` aus. Die zweite Zahl ist negativ, weil **`s`** auf dem 16-Bit-Typ `short` basiert und das 16. Bit – das höchstwertige Bit in `short` – auf 1 gesetzt wird. Da dieses Bit das Vorzeichen darstellt, wird in **`s`** eine Bitfolge gespeichert, die eine negative Zahl darstellt.
`BOOST_BINARY` stellt letztendlich nur eine andere Möglichkeit dar, Zahlen anzugeben. Da Zahlen in C++ standardmäßig den Typ `int` haben, entspricht auch `BOOST_BINARY` einem `int`. Möchten Sie beispielsweise eine Zahl vom Typ `long` definieren, können Sie auf das Makro `BOOST_BINARY_L` zugreifen. Dies entspricht einer Zahl mit dem Suffix L.

Boost.Utility stellt weitere Makros wie `BOOST_BINARY_U` zur Verfügung, um eine Variable ohne Vorzeichenbit zu initialisieren. Dieses ist wie alle anderen Makros in der Headerdatei `boost/utility/binary.hpp` definiert.

Beispiel 69.7 `boost::string_ref` in Aktion

```
#include <boost/utility/string_ref.hpp>
#include <iostream>

boost::string_ref start_at_boost(boost::string_ref s)
{
  auto idx = s.find("Boost");
  return (idx != boost::string_ref::npos) ? s.substr(idx) : "";
}

int main()
{
  boost::string_ref s = "The Boost C++ Libraries";
  std::cout << start_at_boost(s) << '\n';
}
```

Beispiel 69.7 stellt die Klasse `boost::string_ref` vor. `boost::string_ref` ist eine Referenz auf einen String, die lediglich einen lesenden Zugriff erlaubt. Sie ist in gewisser Weise vergleichbar mit `const std::string&`. `const std::string&` setzt jedoch voraus, dass ein String vom Typ `std::string` existiert. `boost::string_ref` kann auch ohne `std::string` verwendet werden. Der entscheidende Vorteil von `boost::string_ref` ist, dass auf Speicherallokationen verzichtet werden kann, wie sie bei einem vielfältigen Einsatz von `std::string` nötig wären.

Beispiel 69.7 sucht in einem String nach dem Wort „Boost“. Wird es gefunden, wird ein String beginnend mit diesem Wort ausgegeben. Wird „Boost“ nicht gefunden, wird ein leerer String ausgegeben. Der String **s** in der Funktion `main()` erhält jedoch nicht den Typ `std::string`, sondern `boost::string_ref`. Es findet daher keine Speicherallokation mit `new` statt. Es wird keine Kopie erstellt, wie es `std::string` tun würde. Stattdessen verweist **s** direkt auf die Zeichenkette „The Boost C++ Libraries“.

Der Typ des Rückgabewerts von `start_at_boost()` lautet ebenfalls nicht `std::string`, sondern `boost::string_ref`. Die Funktion gibt keinen neuen String zurück, sondern eine Referenz. Der von `start_at_boost()` zurückgegebene Wert ist entweder ein Substring des Parameters, der an die Funktion übergeben wurde, oder eine leere Zeichenkette. Die Funktionsweise von `start_at_boost()` setzt voraus, dass der ursprüngliche String mindestens so lange gültig ist wie die Referenzen vom Typ `boost::string_ref`, die auf ihn verweisen. Ist dies wie im obigen Beispiel gewährleistet, können Speicherallokationen, wie sie von `std::string` typischerweise vorgenommen werden, vermieden werden.

Boost.Utility stellt einige weitere Hilfsmittel zur Verfügung, auf die in diesem Buch nicht näher eingegangen wird. Es handelt sich hierbei vorwiegend um Hilfsmittel, die sich entweder speziell an Entwickler von Boost-Bibliotheken richten oder in der Template-Metaprogrammierung zum Einsatz kommen. Die Dokumentation von Boost.Utility kann Ihnen einen Überblick über diese weiteren Hilfsmittel verschaffen.

Kapitel 70

Boost.Assign

Die Bibliothek Boost.Assign bietet Hilfsfunktionen an, um Container zu initialisieren oder Elemente Containern hinzuzufügen. Die Funktionen sind besonders hilfreich, wenn sehr viele Elemente in einem Container gespeichert werden sollen. So ist es dank der von Boost.Assign angebotenen Funktionen nicht notwendig, selbst wiederholt eine Methode wie `push_back()` aufzurufen und Elemente einzeln in einem Container ablegen zu müssen.
Arbeiten Sie in einer Entwicklungsumgebung, die C++11 unterstützt, profitieren Sie von Initialisierungslisten. So können Sie dem Konstruktor von Containern üblicherweise beliebig viele Werte übergeben, mit denen Container initialisiert werden. Dank Initialisierungslisten sind Sie in C++11 nicht auf Boost.Assign angewiesen. Boost.Assign bietet jedoch auch Hilfsfunktionen an, um einem existierenden Container mehrere Werte hinzufügen zu können. Diese Hilfsfunktionen können auch in einer C++11-Entwicklungsumgebung nützlich sein.
Im Beispiel 70.1 werden einige Funktionen vorgestellt, mit denen Container initialisiert werden können. Es reicht, die Headerdatei `boost/assign.hpp` einzubinden, um Zugriff auf sämtliche von Boost.Assign definierten Funktionen zu haben.

Beispiel 70.1 Container initialisieren

```
#include <boost/assign.hpp>
#include <boost/tuple/tuple.hpp>
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <utility>

using namespace boost::assign;

int main()
{
  std::vector<int> v = list_of(1)(2)(3);

  std::stack<int> s = list_of(1)(2)(3).to_adapter();

  std::vector<std::pair<std::string, int>> v2 =
    list_of<std::pair<std::string, int>>("a", 1)("b", 2)("c", 3);

  std::map<std::string, int> m =
    map_list_of("a", 1)("b", 2)("c", 3);

  std::vector<boost::tuple<std::string, int, double>> v3 =
    tuple_list_of("a", 1, 9.9)("b", 2, 8.8)("c", 3, 7.7);
}
```

Boost.Assign bietet drei Funktionen an, um Container zu initialisieren. Die wichtigste dieser drei Funktionen ist `boost::assign::list_of()` – mit dieser Funktion arbeiten Sie grundsätzlich. `boost::assign::map_list_of()` verwenden Sie, wenn Sie auf `std::map` zugreifen. `boost::assign::tuple_list_of()` wiederum setzen Sie ein, wenn Sie Tuple in einem Container initialisieren möchten.
Sie müssen nicht auf `boost::assign::map_list_of()` oder `boost::assign::tuple_list_of()` zugreifen. Sie können alle Container mit `boost::assign::list_of()` initialisieren. Greifen Sie jedoch auf

`std::map` oder einen Container mit Tuplen zu, müssen Sie bei `boost::assign::list_of()` einen Template-Parameter angeben, um der Funktion mitzuteilen, wie Werte im Container gespeichert werden. Dieser Template-Parameter entfällt bei `boost::assign::map_list_of()` und `boost::assign::tuple_list_of()`.
Allen drei Funktionen ist gemein, dass sie ein Proxy-Objekt zurückgeben. Das Objekt, das von `boost::assign::list_of()` und den anderen beiden Funktionen zurückgegeben wird, überlädt den Operator `operator()`. Sie können diesen Operator mehrfach aufrufen, um Werte im Container zu speichern. Obwohl Sie dabei auf ein anderes Objekt und nicht auf den Container zugreifen, ändern Sie durch den Proxy den Container.
Wenn Sie Adapter wie `std::stack` initialisieren möchten, müssen Sie für den Proxy die Methode `to_adapter()` aufrufen. Der Proxy greift dann auf die Methode `push()` zu, die von allen Adaptern angeboten wird.
Beachten Sie, dass `boost::assign::tuple_list_of()` ausschließlich Tuple vom Typ `boost::tuple` unterstützt. Sie können die Funktion nicht verwenden, um Container mit Tuplen aus der Standardbibliothek zu initialisieren.

Beispiel 70.2 Containern Werte hinzufügen

```
#include <boost/assign.hpp>
#include <vector>
#include <deque>
#include <set>
#include <queue>

int main()
{
  std::vector<int> v;
  boost::assign::push_back(v)(1)(2)(3);

  std::deque<int> d;
  boost::assign::push_front(d)(1)(2)(3);

  std::set<int> s;
  boost::assign::insert(s)(1)(2)(3);

  std::queue<int> q;
  boost::assign::push(q)(1)(2)(3);
}
```

Boost.Assign bietet mit `boost::assign::push_back()`, `boost::assign::push_front()`, `boost::assign::insert()` und `boost::assign::push()` weitere Funktionen an, die einen Proxy zurückgeben. Sie müssen diesen Funktionen den Container übergeben, dem Sie neue Elemente hinzufügen möchten. Sie greifen dann wie zuvor über den Operator `operator()` auf den Proxy zu und geben die Werte an, die Sie im Container speichern möchten.
Die vier Funktionen `boost::assign::push_back()`, `boost::assign::push_front()`, `boost::assign::insert()` und `boost::assign::push()` heißen so, weil die von ihnen zurückgegebenen Proxys die gleichnamigen Methoden für den Container aufrufen. So werden im Beispiel 70.2 die drei Zahlen 1, 2 und 3 dem Vektor **v** über den Aufruf von `push_back()` hinzugefügt.
Boost.Assign bietet weitere Funktionen an, um einem Container Werte hinzuzufügen. So können Sie zum Beispiel `boost::assign::add_edge()` verwenden, um einen Proxy für Graphen von Boost.Graph zu erhalten.

Kapitel 71

Boost.Swap

Wenn Sie viele Boost-Bibliotheken verwenden und hin und wieder Daten in Objekten mit `std::swap()` tauschen, sollten Sie stattdessen auf `boost::swap()` zugreifen. Diese Funktion wird von Boost.Swap zur Verfügung gestellt. Um sie einsetzen zu können, müssen Sie die Headerdatei `boost/swap.hpp` einbinden.

Beispiel 71.1 `boost::swap()` in Aktion

```
#include <boost/swap.hpp>
#include <boost/array.hpp>
#include <iostream>

int main()
{
  char c1 = 'a';
  char c2 = 'b';

  boost::swap(c1, c2);

  std::cout << c1 << c2 << '\n';

  boost::array<int, 1> a1{{1}};
  boost::array<int, 1> a2{{2}};

  boost::swap(a1, a2);

  std::cout << a1[0] << a2[0] << '\n';
}
```

`boost::swap()` macht nichts anderes als `std::swap()`. Da viele Boost-Bibliotheken jedoch Spezialisierungen anbieten, um einen Austausch von Daten schneller durchzuführen, und diese Spezialisierungen im Namensraum `boost` definiert sind, kann mit `boost::swap()` von ihnen profitiert werden. So wird im Beispiel 71.1 im ersten Fall innerhalb von `boost::swap()` auf `std::swap()` zugegriffen, um die Werte der beiden `char`-Variablen auszutauschen. Im zweiten Fall wird die von Boost.Array spezialisierte Funktion `boost::swap()` verwendet, die für einen Austausch von Daten in Arrays vom Typ `boost::array` optimiert ist.
Beispiel 71.1 gibt `ba` und `21` aus.

Kapitel 72

Boost.Operators

Boost.Operators stellt zahlreiche Klassen zur Verfügung, mit denen Operatoren automatisch überladen werden können. So wird im Beispiel 72.1 einer Klasse automatisch ein Größer-als-Operator hinzugefügt, weil dieser mit Hilfe des in der Klasse definierten Kleiner-als-Operators implementiert werden kann.

Beispiel 72.1 Größer-als-Operator mit `boost::less_than_comparable` definieren

```
#include <boost/operators.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : public boost::less_than_comparable<animal>
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}

  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"spider", 8};

  std::cout << std::boolalpha << (a2 > a1) << '\n';
}
```

Um Operatoren einer Klasse automatisch hinzuzufügen, muss diese von Klassen abgeleitet werden, die Boost.Operators in der Headerdatei `boost/operators.hpp` zur Verfügung stellt. Wenn eine Klasse wie im Beispiel 72.1 von `boost::less_than_comparable` abgeleitet wird, werden automatisch die Operatoren `operator>`, `operator<=` und `operator>=` definiert.

Ein automatisches Überladen von Operatoren ist möglich, weil viele Operatoren durch andere Operatoren ausgedrückt werden können. So implementiert `boost::less_than_comparable` zum Beispiel den Größer-als-Operator als das Gegenteil des Kleiner-als-Operators: Wenn ein Objekt nicht kleiner als ein anderes ist, muss es größer sein.

`boost::less_than_comparable` setzt voraus, dass es keine gleichen Objekte gibt. Wenn es gleiche Objekte geben kann, darf nicht von `boost::less_than_comparable` abgeleitet werden. In diesem Fall muss `boost::partially_ordered` als Elternklasse verwendet werden. Dazu muss der Operator `operator==` definiert werden, so dass `boost::partially_ordered` feststellen kann, ob nicht kleiner größer oder gleich bedeutet.

Neben `boost::less_than_comparable` und `boost::partially_ordered` werden zahlreiche weitere Klassen angeboten, um nicht nur Vergleichsoperatoren automatisch zu überladen, sondern auch arithmetische und logische Operatoren. Darüber hinaus stehen Klassen zur Verfügung, um Operatoren zu überladen, wie sie üblicherweise von Iteratoren, Zeigern oder Arrays angeboten werden. Da ein automatisches Überladen immer nur möglich ist, wenn andere Operatoren bereits definiert sind, hängt es von Fall zu Fall ab, welche Operatoren Sie vorgeben müssen. Hier hilft nur ein Blick in die Dokumentation.

Index

B

C

D

G

H

I

M

N

O

P

Q

R

S

X

Y

Z