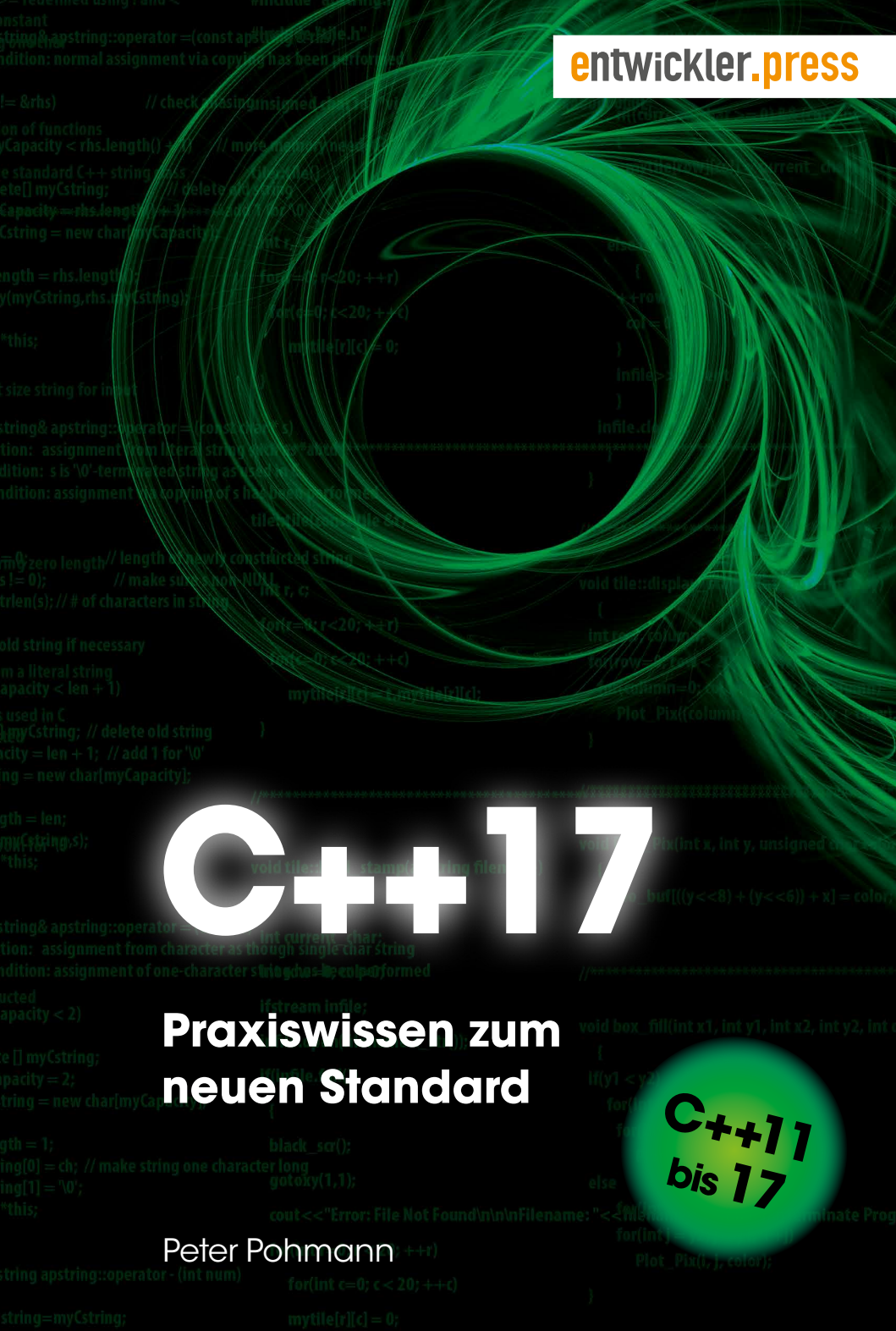
entwickler.press
C++17
Praxiswissen zum neuen Standard
C++11 bis 17
Peter Pohmann

Peter Pohmann

C++17

Praxiswissen zum neuen Standard. Von C++11 bis 17

Peter Pohmann
C++ 17. Praxiswissen zum neuen Standard. Von C++11 bis 17

ISBN: 978-3-86802-361-9

Ein Imprint der Software & Support Media GmbH

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

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Media GmbH
entwickler.press
Schwedlerstraße 8
60314 Frankfurt am Main
Tel.: +49 (0)69 630089-0
Fax: +49 (0)69 630089-89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat/Korrektorat: Björn Bohn, Martina Raschke
Copy-Editor: Nicole Bechtel
Satz: Sibel Sarli
Umschlaggestaltung: Maria Rudi
Titelbild: © Raycat | istockphoto.com
Belichtung, Druck und Bindung: Media-Print Informationstechnologie GmbH, Paderborn

Inhaltsverzeichnis

1 Einleitung

1.1 Ist dieses Buch für Sie?

Als C++-Entwickler haben Sie natürlich mitbekommen, dass es seit 2011 ein runderneuertes „modernes“ C++ gibt, das sich in vielen Konzepten vom klassischen C++ unterscheidet und das seitdem regelmäßig weiterentwickelt wird, mit einem kleinen Update im Jahr 2014 und mit einer größeren Updates im Jahr 2017 (zum Zeitpunkt des Buchs so geplant).

Aber es geht Ihnen vielleicht wie den meisten: Sie stecken im Tagesgeschäft, haben Deadlines zu beachten und unvorhersehbare Fixes zu liefern. Bis jetzt blieb Ihnen keine Zeit, um sich die Neuerungen genauer anzusehen oder gar darüber nachzudenken, wie Sie sie in Ihrer Arbeit einsetzen wollen. Die vorhandenen Bücher zum modernen C++ sind alle recht umfangreich, erheblich umfangreicher jedenfalls als das ursprüngliche C++-Buch „Die C++-Programmiersprache“ von Bjarne Stroustrup aus dem Jahr 1985. Das alles zu lesen, zu verstehen sowie die Vor- und Nachteile abzuwägen, dafür hat die Zeit bisher einfach nicht gereicht.

Wenn diese Beschreibung auf Sie zutrifft, dann ist dieses Buch genau das richtige für Sie. Es strebt nicht danach, möglichst alle Aspekte des neuen Standards ausführlichst zu beschreiben, sondern gibt Ihnen so knapp wie möglich das nötige Wissen und Verständnis an die Hand, die neuen Möglichkeiten sofort einzusetzen. Es beginnt bei den Erläuterungen nicht bei Adam und Eva, sondern setzt voraus, dass Sie über ein solides Praxiswissen mit klassischem C++ verfügen. Andererseits erwartet es von Ihnen nicht, dass Sie die theoretischen Grundlagen von Programmiersprachen noch auswendig aufsagen können. Ein großer Wert liegt darauf, zu jeder Neuerung zu erläutern, wie und wann man sie einsetzen kann. Ab und zu steht hier auch einmal, dass Sie das eher nicht brauchen werden oder lieber die Finger davon lassen sollten.

Vom Aufbau her stehen die einzelnen Kapitel, von denen jedes genau ein neues Feature beschreibt, weitgehend für sich. So brauchen Sie dieses Buch nicht unbedingt von vorne bis hinten zu lesen, sondern können sich auch einfach die Themen heraussuchen, die Ihnen am interessantesten erscheinen. Weil natürlich manche Neuigkeiten auf anderen aufbauen, finden Sie an den entsprechenden Stellen die Verweise auf die nötigen Voraussetzungen.

1.2 Aufbau

Die Kapitel sind locker zu vier Themenbereichen zusammengefasst. Im ersten Teil „Sprache“ geht es um Erweiterungen an der Programmiersprache selbst. Hier werden hauptsächlich neue Schlüsselwörter oder neue Bedeutungen vorhandener Schlüsselwörter vorgestellt.

Anschließend kommt ein Teil über „Templates“, den Sie komplett überspringen können, wenn Sie keine Templates bei Ihrer Arbeit einsetzen. Hier sind sowohl Sprach- als auch Bibliothekserweiterungen zusammengefasst, die das Programmieren mit Templates betreffen. Früher oder später sollten Sie sich übrigens unbedingt mit diesem Themengebiet auseinandersetzen. Das Arbeiten mit Templates ist gerade durch die Erweiterungen seit C++11 erheblich angenehmer geworden. Dieses Buch will allerdings keine Einführung in die Templateprogrammierung sein.

Im dritten Teil geht es um die Erweiterungen der Standardbibliothek. Er ist mit „Bibliothek“ überschrieben und enthält Dutzende von neuen Funktionen und Klassen von ganz einfachen Dingen wie den Funktionen *next* und *prev* bis zu umfangreichen Bibliotheken wie der für Zufallszahlen oder Zeitpunkte und Zeitdauern.

Alles, was mit Multithreading und paralleler Programmierung zu tun hat, fasst der vierte Teil unter „Nebenläufigkeit“ zusammen. Auch hier kommen sowohl Spracherweiterungen als auch neue Bibliotheksklassen vor, weil sie zusammen ein sinnvolles Ganzes ergeben. Wenn diesem Ihr Hauptinteresse gilt, fangen Sie einfach damit an und folgen den Verweisen für die wenigen Spracherweiterungen und Bibliotheksklassen, die als Grundlage benötigt werden.

Im Anhang finden Sie ein Glossar, welches einige Fachbegriffe rund um C++ erläutert. Diese Begriffe gehören zwar nicht zu den Neuerungen seit C++11. Sie tauchen aber im Hauptteil auf und sind dann doch so speziell, dass sie nicht jeder sofort parat haben kann. Immer, wenn Sie den Pfeil vor einem Wort sehen wie in →trivialer Typ, dann wissen Sie, dass es dazu einen Glossareintrag gibt.

Die Beschreibung jedes Sprachmerkmals hat einen sich wiederholenden Aufbau. Zuerst kommt unter „Worum geht es?“ eine Kurzfassung, die in wenigen Sätzen erklärt, was das Feature tut. Darauf folgt unter der Überschrift „Was steckt dahinter?“ eine Motivation und Erläuterung, wo erklärt wird, wieso das Feature eingeführt wurde und wie es funktioniert. Als drittes kommen Tipps, wie Sie die Erweiterung am besten einsetzen, und wie nicht. Dieser Abschnitt ist mit „Wie setzt man es ein?“ überschrieben. In „Wer unterstützt es?“ folgen Hinweise zur Compilerunterstützung in Visual C++ und GCC. Abschließend runden Verweise auf den Header und gegebenenfalls weiterführende Literatur die Beschreibung ab.

Damit Sie die für Sie interessanten Neuigkeiten möglichst schnell finden, gibt es neben einem umfangreichen Stichwortverzeichnis gleich anschließend eine Feature-Tabelle als Wegweiser. Hier finden Sie alle Erweiterungen von C++ mit einer Bewertung bezüglich Relevanz, Expertentum und Spezialwissen mit Verweis auf das jeweilige Kapitel.

1.3 Compiler

Bei den Compilern beschränkt sich dieses Buch auf die beiden meistverwendeten, nämlich Visual C++ unter Windows und GNU unter Linux. Viele Beispiele wurden sowohl mit MSVC 2015 Update 3 als auch GCC 5.4 bzw. 7.0 getestet. Manchmal war das aber nicht möglich, weil insbesondere MSVC 2015 bei den Features von C++17 noch große Lücken hat. Die Version 2017 war noch nicht veröffentlicht, als dieses Buch geschrieben wurde.

Insbesondere für GCC ist auch nicht vollständig dokumentiert, ab welcher Version eine bestimmte Erweiterung der Standardbibliothek unterstützt wird. Deshalb finden Sie bei der Beschreibung der Compilerunterstützung öfter Formulierungen wie „spätestens ab 4.8“. Das bedeutet dann,

dass laut Dokumentation und eigenen Tests das Feature in GCC 4.8 zur Verfügung steht, dass aber nicht bekannt ist, seit wann es schon existiert.

1.4 Schreibweisen

Dieses Buch wurde in der Originalfassung auf Deutsch geschrieben und bemüht sich deshalb, wo immer sinnvoll möglich, deutsche oder eingedeutschte Begriffe zu nutzen. Dies führt zwangsläufig zu einigen schwierigen Fällen. Zum Beispiel könnte man Compiler als Übersetzer verwenden, aber genau genommen besteht der Übersetzer aus Compiler und Linker zusammen. Statt Templates könnte man Schablonen sagen, aber dieser Begriff hat sich nicht genug eingebürgert, um ihn zu verwenden. Das Wort *variadisch* steht nicht im Duden, allerdings steht auch seine englische Entsprechung *variadic* nicht im Oxford Dictionary, so dass es hier verwendet wird (für *variadic templates*). Es gibt noch viele weitere Beispiele für Zweifelsfälle, sie wurden so gut gelöst, wie es eben möglich ist.

Generell wird in Programmierrichtlinien und Coding-Style-Guides empfohlen, auf die using-namespace-Anweisung zu verzichten, und gerade bei Bezeichnern aus der Standardbibliothek das *std* immer mit dazuzuschreiben. Diese Regel wird in diesem Buch grundsätzlich beachtet, an einigen Stellen aber aus Platzgründen übergangen. In den Codebeispielen passen ca. 70 Zeichen in eine Zeile und Umbrüche führen zu sehr schlechter Lesbarkeit. Deshalb wurden lange Zeilen, wo immer möglich, gekürzt, unter anderem durch Weglassen von *std::*, durch Einsatz kürzerer Bezeichner, durch kürzere Literale für die Ausgabe, durch Einführung zusätzlicher Variablen usw. Diese Maßnahmen würden in echtem Code die Lesbarkeit verschlechtern und werden deshalb nicht empfohlen, im Buch jedoch verbessern sie die Übersichtlichkeit.

1.5 Dank

Dieses Buch hat der Autor nicht alleine geschaffen. Mein Dank geht an meine Frau Judith und meine Söhne Simon und Philipp, die nicht nur

viele Stunden auf Mann und Vater verzichten mussten, sondern auch auf den leistungsstärksten Computer unserer Familie. Peter Biereder, Jörg Roth und Kurt Holzinger haben sich durch das Manuskript gekämpft und geholfen, Fehler und Ungereimtheiten zu beseitigen. Sebastian Burkart und Martina Raschke vom Verlag haben das Projekt betreut und mir mit Rat und Tat zur Seite gestanden.

1.6 Kontakt

Über Anmerkungen zu diesem Buch freue ich mich unter peter.pohmann@dataweb.de.

1.7 Neue Features im Überblick

Die folgende Tabelle soll Ihnen einen Überblick über die Neuheiten und ihre praktische Bedeutung geben. Jedes neue Feature hat eine Bewertung in den drei Kategorien Relevanzfaktor, Expertenfaktor und Spezialistenfaktor. In jeder Kategorie bekommt das Feature ein bis drei Symbole für gering, mittel und ausgeprägt.

*** Relevanzfaktor: Je mehr Sternchen, desto wichtiger ist das Feature für Ihren Programmiereralltag. Features mit drei Sternchen sollten Sie möglichst sofort lesen und einsetzen, vorausgesetzt, es ist von allgemeiner Bedeutung oder für Ihr Spezialgebiet relevant.

!!! Expertenfaktor: Mehr Ausrufezeichen bedeuten höheren Lernaufwand. Ein Feature mit hohem Expertenfaktor und geringer Relevanz werden Sie sich wohl erst ansehen, wenn Sie einmal Zeit übrig haben oder wenn es für Ihr Spezialgebiet genau passt.

??? Spezialistenfaktor: Manche Features sind nur für spezifische Anwendungsfälle interessant, beispielsweise für Simulation oder systemnahe Programmierung. Auch wenn sie in diesen Anwendungsgebieten sehr wichtig sind, kommen viele Softwareentwickler auch gut ohne sie aus.

Sprache

Feature	Relevanz	Experte	Spezialist	Kapitel
Automatische Typableitung mit *auto*	***	!	?	2.1
Nachgestellter Ergebnistyp	*	!	?	2.2
streng typisierte Aufzählungen	***	!	?	2.3
Ausnahmespezifikation	**	!	?	2.4
Attribute	**	!	?	2.5
rvalue-Referenzen	**	!!	?	2.6
explizites Überschreiben	**	!!	?	2.7
Ableitung und Überschreibung verhindern	***	!	?	2.8
automatische Implementierung löschen und wiederherstellen	**	!!	?	2.9
Konstruktoraufruf in Konstruktoren	**	!	?	2.10
Konstruktoren vererben	**	!	?	2.11
unbeschränkte Unions	*	!!	?	2.12
bereichbasierte *for*-Schleife	***	!	?	2.13
Initialisierer-Listen	***	!!	?	2.14
einheitliche Initialisierung	***	!	?	2.15
Initialisierer für Elementvariablen	***	!	?	2.16
Lambda-Funktionen	***	!	?	2.17
initialisierte Lambda-Captures	**	!!	?	2.18
generische Lambda-Funktionen	*	!!	?	2.19
konstante Ausdrücke	***	!	?	2.20
erweiterte *Friend*-Deklaration	**	!!	?	2.21
binäre Literale	*	!	???	2.22
Zahlentrennzeichen	*	!	?	2.23
Zeichenketten-Literale	**	!	?	2.24
benutzerdefinierte Literale	*	!	??	2.25
nullptr	***	!	?	2.26
Inline-Namensräume	*	!!	??	2.27

Feature	Relevanz	Experte	Spezialist	Kapitel
statische Zusicherungen	**	!	?	2.28
expliziter Typkonvertierungsoperator	**	!	?	2.29
alignof und *alignas*	*	!	?	2.30
sizeof für Elementvariable	*	!!	???	2.31
strukturierte Bindung	**	!	?	2.32
Bedingung mit Initialisierer	**	!	?	2.33
geschachtelte Namensraumdefinition	**	!	?	2.34
Inline-Variable	**	!	?	2.35

Templates

Feature	Relevanz	Experte	Spezialist	Kapitel
Variablentemplates	*	!!	?	3.1
Typberechnung	**	!!	?	3.2
automatische Typableitung mit *declspec(auto)*	*	!!	??	3.3
unbenannte und lokale Typen als Templateargumente	*	!	?	3.4
Default-Argumente für Funktionstemplates	*	!	?	3.5
abgeleiteter Templateparametertyp	*	!!	??	3.6
Typ-Alias	*	!	??	3.7
externe Templateinstanziierung	**	!!	??	3.8
variadische Templates	***	!!	?	3.9
Faltungsausdrücke	**	!!	??	3.10
Referenz-Wrapper	**	!	?	3.11
Typmerkmale	**	!	?	3.12
forward-Funktion	**	!!!	??	3.13
declval-Funktion	*	!!	??	3.14
bedingte Kompilierung im Template	**	!!	?	3.15

Bibliothek

Feature	Relevanz	Experte	Spezialist	Kapitel
integrale Typen mit definierter Länge	**	!	?	4.1
unique_ptr-Klasse	***	!	?	4.2
shared_ptr-Klasse	**	!!	?	4.3
weak_ptr-Klasse	**	!!	?	4.4
move-Funktion	**	!	?	4.5
bind-Funktion	**	!!	?	4.6
function-Klasse	**	!!	?	4.7
einfach verkettete Liste	*	!!	?	4.8
Array mit fester Länge	*	!	?	4.9
Hash-basierte Container	**	!	?	4.10
Tupel	**	!!	??	4.11
Varianten	**	!	?	4.12
beliebige Werte	*	!	??	4.13
optionale Werte	***	!	?	4.14
Zugriff auf Element über seinen Typ	**	!	??	4.15
Brüche	*	!!	??	4.16
Zeitpunkte und Zeitdauern	***	!!	?	4.17
Zeitliterale	**	!	?	4.18
Zufallszahlen	**	!!	???	4.19
Reguläre Ausdrücke	**	!!	??	4.20
system_error-Klasse	*	!!	??	4.21
next- und *prev*-Funktionen	*	!	?	4.22
Containererweiterung	**	!	?	4.23
IO-Manipulator *quoted*	**	!	??	4.24
String-Sichten	***	!	?	4.25
Konvertierung von Zeichenketten in Zahlen	***	!	?	4.26

Nebenläufigkeit

Feature	Relevanz	Experte	Spezialist	Kapitel
async-Funktion	***	!	?	5.1
Threads	**	!!	??	5.2
packaged_task-Klasse	*	!!	??	5.3
promise-Klasse	**	!!	??	5.4
shared_future-Klasse	*	!!	???	5.5
Mutexe	***	!	?	5.6
zweistufige Mutexe	**	!!	??	5.7
lock_guard-Klasse	***	!	?	5.8
rekursiver Mutex	***	!	?	5.9
lock- und *try_lock*-Methoden	**	!!	?	5.10
unique_lock-Klasse	**	!	?	5.11
Thread-lokale Daten	**	!	?	5.12
einmalige Ausführung	**	!	?	5.13
Bedingungsvariablen	**	!!	?	5.14
atomare Operationen	**	!!	???	5.15
exception_ptr-Klasse	**	!	??	5.16

2 Sprache

2.1 Automatische Typableitung mit „auto"

Worum geht es?

Wenn Sie den Typ einer zu definierenden Variablen als *auto* angeben, bestimmt der Compiler den tatsächlichen Typ automatisch aus dem Initialisierer. Zum Beispiel:

```
auto pi = 3.1415;

std::string s("Anthony");
auto l = s.length();

std::vector<int> v;
auto i = v.begin();
```

Seit C++14 funktioniert das auch mit dem Ergebnistyp einer Funktion:

```
auto Calculate(double a, int b, float c) {
  return b/a * c;
}
```

Was steckt dahinter?

Den korrekten Typ für eine Variable anzugeben, ist in manchen Fällen eine Qual. Nehmen Sie zum Beispiel folgende Definition einer *map*, die zu einem Bezeichner in einem konfigurierbaren Zeichensatz eine Funktion liefert, welche zwei *double*-Argumente erwartet und eine komplexe Zahl zurückgibt.

```
std::map<
  std::basic_string<uchar>,
  std::complex<float> (*)(double, double)
> functions;
```

Nun soll die *map* in einer *for*-Schleife abgearbeitet werden. Wie war nochmal gleich der genaue Typ des Iterators?

```
for (std::map<std::basic_string<uchar>,
  std::complex<float> (*)(double, double)>::const_iterator it =
    functions.begin(); it != functions.end(); ++it) {
  …
}
```

Das einzutippen macht wenig Spaß und fördert auch nicht gerade die Lesbarkeit des Codes. Hier kommt im neuen Standard das Schlüsselwort *auto* zur Hilfe:

```
for (auto it = functions.begin(); it != functions.end(); ++it) {
  …
}
```

Das einfache Prinzip: *„auto* definiert die Variable mit dem Typ des Initialisierers“ funktioniert aber nicht überall intuitiv. Die Typableitung für *auto* unterliegt denselben Regeln wie die für Templateparameter. So gilt für Referenzen:

Wenn der Initialisierer eine Referenz liefert, definiert *auto* den unreferenzierten Typ.

Nach

```
class Foo {
public:
  const std::string& GetName() const;
  std::string* CreateDescription() const;
  …
};

Foo foo;
auto name = foo.GetName();
```

hat *name* also den Typ *string* und nicht *const string&*.

Sie können aber auch explizit definieren, dass der automatische Typ konstant, eine Referenz oder ein Zeiger sein soll:

```
const auto& name = foo.GetName();
```

macht *name* zu einer konstanten *string*-Referenz.

Als Zeiger können Sie den automatischen Typ nur dann deklarieren, wenn der Initialisierer auch eine Adresse liefert. Eine Konvertierung ist nicht vorgesehen. Technisch gesehen ist also

```
auto* desc = foo.CreateDescription();
```

identisch mit

```
auto desc = foo.CreateDescription();
```

Ersteres macht aber deutlicher, dass es sich bei *desc* um einen Zeiger handelt.

Auf eine Funktion angewendet bestimmt sich der Ergebnistyp dadurch, dass die beschriebenen Regeln auf den zurückgegebenen Ausdruck angewendet werden. Der Ergebnistyp der Funktion *Calculate* ist also derselbe, der für *r* in folgender Umformung berechnet werden würde:

```
auto r = b/a * c;
return r;
```

Das Schlüsselwort *auto* wird auch für Rückgabewerte von Funktionen im Zusammenhang mit dem nachgestellten Typ benutzt, wie im Kapitel 2.2 beschrieben wird.

Vielleicht erinnern Sie sich, dass das Schlüsselwort *auto* früher dafür verwendet wurde, eine Stack-Variable (automatic variable) zu definieren. Das ist schon lange nicht mehr nötig und ab C++11 auch nicht mehr möglich, weil das Schlüsselwort eine neue Bedeutung bekommen hat.

Wie setzt man es ein?

Es ist verführerisch, bei Variablendefinition einfach nur *auto* zu schreiben und den Rest der Arbeit dem Compiler zu überlassen. Namhafte Autoren empfehlen aus guten Gründen, *auto* einzusetzen, wo es möglich ist (z. B. Meyers 2015). Fehler durch falsche Typdeklaration und durch ungewollte Konvertierungen werden so vermieden. Außerdem passt sich der automatische Typ an, wenn der Typ des Initialisierers durch eine Programmänderung nicht mehr derselbe ist.

Andererseits gibt es auch Fälle, wo bei der Definition mit *auto* der resultierende Typ nicht der ist, den Sie intuitiv explizit geschrieben hätten. Hier sind ein paar Beispiele, bei denen Sie vorsichtig sein müssen:

```
std::string n = "Alexandrescu";
auto n = "Alexandrescu"; // n ist jetzt vom Typ const char*

BaseClass* o = new DerivedClass();
auto* o = new DerivedClass(); // o ist jetzt von Typ DerivedClass*

const std::string* d = foo.CreateDescription();
auto d = foo.CreateDescription(); // d ist nicht mehr const
```

Problematisch ist auch die Verständlichkeit des Codes, wenn der Typ einer Variablen nicht mehr offensichtlich ist. Sehen Sie sich dieses Beispiel an:

```
auto x = foo1->GetX();
auto y = foo1->GetFactor();
auto r = foo2->Process(x, y);
auto z = r - x;
```

Preisfrage: Welchen Typ hat *z*?

Am besten setzt man *auto* da ein, wo eine Variable oder Funktion mit einem technisch komplizierten Typ definiert wird, der zum Verständnis des Ablaufs aber weiter nichts beiträgt. Beispiele dafür sind Iteratoren, wie oben gezeigt, oder auch Lambda-Funktionen (siehe 2.17) und andere → aufrufbare Objekte sowie Templateklassen.

```
int Func(double, std::string, double);

auto lambda = [&](double d, const std::string& s) {
  return Func(d, s, d);
};
auto f = std::bind(Func, std::placeholders::_1, "Williams",
  std::placeholders::_2);

std::map<int, std::string> m;
auto e = m[12];
```

Wo der Typ der Variablen aus der Initialisierung nicht eindeutig hervorgeht, aber für das weitere Verständnis des Codes wichtig ist, sollten Sie überlegen, bei der expliziten Deklaration zu bleiben. Dies gilt erst recht für Funktionen, bei denen man die Implementierung lesen muss, um den Ergebnistyp zu erschließen.

Wer unterstützt es?

Standard C++11, für Ergebnistypen C++14

MSVC ab 2010, für Ergebnistypen ab 2015

GCC ab 4.4, für Ergebnistypen ab 4.9

2.2 Nachgestellter Ergebnistyp

Worum geht es?

Statt

```
int Func(double d);
```

können Sie jetzt auch schreiben

```
auto Func(double d)->int;
```

Was steckt dahinter?

Die neue Schreibweise vereinfacht an manchen Stellen die Angabe des Ergebnistyps. Sehen Sie sich zum Beispiel diese Klasse an:

```
class Foo {
public:
  enum Color { red, green, blue };
  Color GetColor() const;
private:
  Color color_;
};
```

Die Implementierung von *GetColor* mussten Sie bisher so schreiben:

```
Foo::Color Foo::GetColor() const {
  return color_;
}
```

Nun können Sie es auch folgendermaßen formulieren:

```
auto Foo::GetColor() const -> Color {
  return color_;
}
```

Der nachgestellte Ergebnistyp befindet sich schon innerhalb des Sichtbarkeitsbereichs der Klassenelemente. Das gilt für den vorangestellten nicht. Deshalb können Sie im zweiten Fall auf den zusätzlichen Klassennamen verzichten und ersparen sich in manchen Fällen einige Schreibarbeit.

Wie setzt man es ein?

Der nachgestellte Ergebnistyp bricht mit lange eingeprägten Denkmustern beim Lesen von Quelltext. Sein Einsatz ist also nur in Sonderfällen zu empfehlen.

Einen echten Gewinn bringt die neue Schreibweise im Zusammenspiel mit der berechneten Typdeklaration mit *decltype*, die im Kapitel 3.2 beschrieben wird. Außerdem wird sie für Lambda-Funktionen benutzt (siehe Kapitel 2.17).

Wer unterstützt es?

Standard C++11

MSVC ab 2010

GCC ab 4.4

2.3 Streng typisierte Aufzählungen

Worum geht es?

Wenn Sie Ihre Aufzählungen mit dem zusätzlichen Schlüsselwort *struct* oder *class* kennzeichnen, dürfen die Werte nur noch mit vorangestelltem Aufzählungsnamen angesprochen werden und können nicht mehr versehentlich nach *int* konvertiert werden:

```
// Definiert Linienstile durchgezogen, gepunktet, gestrichelt
enum struct LineStyle { solid, dotted, dashed };

void DrawLine(LineStyle style);

DrawLine(LineStyle::dotted);
```

Statt *enum struct* können Sie auch mit genau derselben Bedeutung *enum class* schreiben.

Was steckt dahinter?

Die Wertbezeichner in einem herkömmlichen *enum* sind in seinem Namensraum global definiert. Da diese Bezeichner aber meistens „schöne" Namen haben, möchte man sie nicht auf diese Weise „verschwenden" und den Namensraum verschmutzen. Daher sehen die meisten Programmierrichtlinien vor, dem Wertbezeichner ein Präfix voranzustellen. Zum Beispiel so:

```
enum LineStyle { lsSolid, lsDotted, lsDashed };
```

So richtig gut lesbar ist das nicht, vor allem, wenn weitere Aufzählungen dazu kommen:

```
enum FillStyle { fsSolid, fsHatched, fsHollow };
DrawCircle(lsSolid, fsSolid);
```

Da sieht es mit den neuen C++-Aufzählungen doch gleich viel lesbarer aus:

```
enum struct LineStyle { solid, dotted, dashed };
enum struct FillStyle { solid, hatched, hollow };
DrawCircle(LineStyle::solid, FillStyle::solid);
```

Das ist aber noch nicht alles. Auch wenn die neuen Aufzählungstypen standardmäßig immer noch auf *int*-Werten basieren, erlauben sie doch keine implizite Konvertierung mehr. Dadurch werden Fehler eher vom Compiler erkannt und Sie brauchen nicht mehr typischen Code wie diesen hier zu schreiben:

```
void DrawCircle(LineStyle lineStyle, FillStyle fillStyle){
  assert(lineStyle >= lsSolid && lineStyle <= lsDashed);
  assert(fillStyle >= fsSolid && fillStyle <= fsDashed);
  …
}
```

Mit den streng typisierten Aufzählungen dürfen Sie davon ausgehen, dass keine illegalen Werte übergeben oder zugewiesen werden können, so wie Sie ja bei einem *short* auch nicht prüfen müssen, ob er nicht etwa größer als 32 767 ist.

```
LineStyle lineStyle;
lineStyle = 1; // Compilerfehler
lineStyle = LineStyle::dotted + 1; // Compilerfehler
```

Natürlich können Sie auch beim neuen Aufzählungstyp die nummerischen Werte festlegen. C++11 geht sogar noch einen Schritt weiter und erlaubt die Definition des Basistyps. Dadurch können Sie die Speichergröße des *enum* selbst bestimmen, wenn Sie dabei die neuen Integer-Typen wie *uint8_t* (siehe Kapitel 4.1) benutzen, sogar für alle Plattformen auf die identische Weise.

```
enum struct LineStyle: uint8_t {
  solid = 0,
  dotted = 1,
  dashed = 2
};
```

Im Unterschied zu klassischen Aufzählungstypen können *enum structs* auch vorwärts deklariert werden:

```
enum struct LineStyle: uint8_t;
enum struct FillStyle: uint8_t;

void DrawCircle(LineStyle lineStyle, FillStyle fillStyle);

enum struct LineStyle: uint8_t {
  solid = 0,
  dotted = 1,
  dashed = 2
};
```

Wie setzt man es ein?

enum structs und *enum classes* sind die besseren Aufzählungstypen. Sie sollten konsequent eingesetzt werden, sofern keine Kompatibilität mit Code nach altem Standard gefordert ist. Das fördert die Lesbarkeit und vermeidet Fehler durch Zuweisung ungültiger Werte.

Durch Festlegung eines kleineren zugrundeliegenden Typs können Sie Speicherplatz sparen. Oft macht das keinen wesentlichen Unterschied, aber in einem Vektor von 1 000 000 Linienarten kann es sich schon lohnen.

Grundsätzlich sind bei der Wahl des Basistyps die neuen Integer-Typen mit fester Länge zu empfehlen. Schließlich ändert sich die Anzahl der Aufzählungswerte nicht zwischen den Plattformen, also sollte auch der Basistyp einen identischen Wertebereich haben. Dies hilft dann wiederum bei der Serialisierung und Deserialisierung.

Größere Basistypen als *int* brauchen Sie nicht explizit anzugeben, der Compiler wählt ihn geeignet.

Wenn Sie den Aufzählungstyp für Flags benutzen möchten, also einzelne Werte zu einem Gesamtwert kombinieren, dann ist es sinnvoll, den bitweisen *Oder*-Operator zu definieren:

```
enum struct AggregationFlags: uint32_t {
  minimum = 0x000001;
  maximum = 0x000002;
  average = 0x000004;
  first   = 0x000008;
  last    = 0x000010;
};

AggregationFlags operator|(AggregationFlags a,
                                        AggregationFlags b) {
  typedef std::underlying_type<AggregationFlags>::type
                                                    EnumType;
  return static_cast<AggregationFlags>(
    static_cast<EnumType>(a) | static_cast<EnumType>(b));
}
```

Hier ist *underlying_type* eine Typeigenschaft, mit der sich bestimmen lässt, auf welchem Basistyp die Aufzählung beruht. Typeigenschaften und andere Typmerkmale sind das Thema des Kapitels 3.12.

Ob Sie *enum struct* oder *enum class* benutzen, macht technisch gesehen keinen Unterschied. Man könnte allerdings argumentieren, dass der Unterschied zwischen *struct* und *class* an anderen Stellen darin liegt, dass die Elemente von *struct* standardmäßig öffentlich sind, während sie bei *class* privat sind. Da die Elemente einer *enum struct* oder *enum class* immer öffentlich sind, liegt die Bezeichnung *enum struct* näher.

Wer unterstützt es?

Standard C++11

MSVC ab 2012

GCC ab 4.4

2.4 Ausnahmespezifikation

Worum geht es?

Die Ausnahmespezifikation mit *throw* wurde als veraltet (*deprecated*) eingestuft:

```
void Foo(int a) throw(logical_error); // Nicht mehr benutzen.
```

Teilweise wird sie ersetzt durch das neue Schlüsselwort *noexcept*. Es bedeutet, dass die Funktion keine Ausnahme wirft, und kann zur Übersetzungszeit ausgewertet werden:

```
void Bar(int a) noexcept;
```

Was steckt dahinter?

Die *throw*-Spezifikationen wurden von der Entwicklergemeinde nie richtig angenommen und werden schon länger von führenden Köpfen wie Herb Sutter (Sutter 2005) nicht mehr empfohlen. Die Idee war, dass eine Funktion dokumentiert, welche Ausnahmen sie wirft. Die Erfahrung zeigt aber, dass das für den Entwickler eine schwer lösbare Aufgabe ist, vor allem, wenn Templates oder Ableitungshierarchien im Spiel sind. Der Compiler kann die Spezifikation zur Übersetzungszeit nicht überprüfen. Wenn dann zur Laufzeit eine unerwartete *Exception*-Klasse geworfen wird, beendet sich das Programm sofort. Dadurch führen Ausnahmespezifikationen oft eher zu schlechteren Programmen als zu stabileren.

Die Grundidee ist aber nach wie vor interessant und deshalb wurde in C++11 eine neue Form der Spezifikation eingeführt, die als Ersatz für das bisherige *throw()* dient. Sie spezifiziert nur das, was auch im Entwurf schon festgelegt werden sollte, nämlich, ob eine Funktion überhaupt eine Ausnahme werfen kann oder nicht. Wenn sie mit *noexcept* gekennzeichnet ist und zur Laufzeit doch eine Exception auslöst, wird das Programm sofort mit *terminate* beendet.

Der Zweck des neuen Schlüsselwortes liegt ausschließlich in der Dokumentation. Auch die neue Variante kann vom Compiler nicht verifiziert werden. Allerdings kann der Compiler die Information von *noexcept* verwenden, um den erzeugten Code zu optimieren.

Bis hierher sieht es so aus, also ob *noexcept* einfach nur eine Reduktion von *throw* auf ein sinnvolles Maß darstellt. Es hält aber auch eine Erweiterung bereit, die für den Einsatz mit Templatefunktion notwendig ist. Hier hängt ja die Frage, ob die Funktion eine Ausnahme wirft oder nicht, vom Templateargument ab. Zum Beispiel wird bei

```
vector<MyClass> myVector;
myVector.push_back(MyObject(..));
```

ein Zuweisungsoperator für *MyObject* aufgerufen, der je nach Implementierung von *MyObject noexcept* sein könnte oder nicht. Diese Abhängigkeit können Sie explizit im *noexcept*-Spezifizierer angeben:

```
template<typename T>
void DoSomething(T a, T b) noexcept(std::is_trivial<T>::value) {
  …
}
```

Der Spezifizierer *noexcept* hat hier also ein Argument, das angibt, ob die *noexcept*-Zusicherung gilt oder nicht. Die Funktion *DoSomething<T>* ist genau dann *noexcept*, wenn *T* ein trivialer Typ ist. (*is_trivial* ist eine Typeigenschaft, auf die im Kapitel 3.12 eingegangen wird.)

Für den Fall, dass in *DoSomething* eine Funktion *Foo* von *T* aufgerufen wird, können Sie das *noexcept* von *DoSomething* auch vom *noexcept*-Spezifizierer von *Foo* abhängig machen:

```
template<typename T>
void DoSomething(T a) noexcept(noexcept(a.Foo())) {
  a.Foo();
  …
}
```

Das innere *noexcept* ist der *noexcept*-Operator. Er liefert dann *false*, wenn sein Argument nicht sicher als *noexcept* identifiziert werden kann.

Wie setzt man es ein?

Es ist sicherlich sinnvoll, Funktionen, die keine Ausnahme werfen (sollten), entsprechend mit *noexcept* zu kennzeichnen. Es hilft nicht nur dem Compiler beim Optimieren, sondern insbesondere dem Leser des Programms beim Verständnis und dem Aufrufer der Funktion bei der Ausnahmebehandlung.

Die Verwendung des *noexcept*-Operators in Templatefunktionen ist die konsequente Fortführung dieses Prinzips, erscheint aber recht kompliziert, verglichen mit den Vorteilen.

Wer unterstützt es?

Standard C++11

MSVC ab 2015

GCC spätestens ab 4.8

2.5 Attribute

Worum geht es?

Attribute sind eine Möglichkeit, dem Compiler zusätzliche Hinweise zu geben, beispielsweise, dass die Funktion nie zurückkehren wird:

```
[[noreturn]]
void throwException() {
  throw "exception";
}
```

Was steckt dahinter?

Die Möglichkeit für solche Hinweise existiert in vielen Programmiersprachen und auch in C++ haben die Compilerhersteller schon lange proprietäre Sprachmittel eingeführt, beispielsweise *__declspec* bei Visual C++ oder *__attribute__* bei GCC. Die Attribute im Standard führen eine

einheitliche Syntax für diese Zwecke ein und definieren gleichzeitig einige konkrete Attribute, die dann auf allen Plattformen dieselbe Bedeutung haben. Es ist allerdings weiterhin zulässig, dass Implementierungen ihre eigenen Attribute definieren.

Attribute können praktisch allen Elementen eines Programms zugeordnet werden: Typen, Variablen, Funktionen, Namen, Codeblöcken, Modulen, Namensräumen und Enumeratoren. Manche Attribute wie *deprecated* erlauben auch Argumente:

```
[[deprecated("Veraltet, stattdessen newFunc benutzen")]]
void oldFunc() {
  …
}
```

Ein weiteres Attribut zeigt an, dass das „Durchfallen“ von einem *case*-Label zum nächsten beabsichtigt ist.

```
switch(n) {
case 1:
  DoFor1();
  break;
case 2:
  if (Ask()) {
    DoFor2a();
    break;
  } else {
    DoFor2b();
    [[fallthrough]];
  }
case 3:
  DoFor3And2b();
  break;
}
```

Das Attribut steht hier an einer seltsamen Stelle, weil es syntaktisch ein Attribut für eine leere Anweisung ist, welche wiederum die letzte einer Sequenz sein muss, bevor ein neues *case*-Label kommt. Wenn das Attribut zum *case*-Label gehören würde, könnte man keine Einzelfälle unterscheiden, wie das oben für *n == 2* geschieht.

Die folgenden Attribute sind im Standard definiert:

Attribut	Bedeutung
noreturn	Funktion kehrt nie zurück
carries_dependency	Funktionsverhalten in Bezug auf die Speicherzugriffsreihenfolge
deprecated *deprecated("Meldung")*	Typ, Funktion etc. soll nicht mehr benutzt werden. Erzeugt eine Compilerwarnung
fallthrough	Keine Compilerwarnung beim „Durchfallen" von einem *case*-Label zum nächsten
nodiscard	Warnung, wenn Typ, Aufzählung oder Funktion in einem Ausdruck benutzt wird, dessen Ergebnis verworfen wird.
maybe_unused	Keine Warnung, wenn das Objekt nicht benutzt wird

Wie setzt man es ein?

Mit *[[noreturn]]* vermeiden Sie unnötige Warnungen des Compilers. *[[deprecated]]* erzeugt beim Übersetzen einen Hinweis auf die Benutzung eines veralteten Konstrukts. *[[fallthrough]]* unterdrückt die Warnung des Compilers, wenn eine Anweisungssequenz nicht mit *break* abgeschlossen wurde. Attribute verbessern Ihren Code, weil sie helfen, Ihre Absicht auszudrücken, und sollten dementsprechend konsequent genutzt werden.

Wer unterstützt es?

- Attribute bei Namensräumen und Enumeratoren
  - Standard C++17
  - MSVC ab 2017
  - GCC ab 4.9 (Namensräume) bzw. 6 (Enumeratoren)
- *fallthrough*, *nodiscard*, *maybe_unused*
  - Standard C++17
  - MSVC ab 2017
  - GCC ab 7
- *deprecated*

  - Standard C++14
  - MSVC ab 2015
  - GCC ab 4.9
- Attribute allgemein, *noreturn*, *carries_dependency*
  - Standard C++11
  - MSVC ab 2015
  - GCC ab 4.8

2.6 „Rvalue"-Referenzen

Worum geht es?

Eine *Rvalue*-Referenz ist eine Referenz, die nur auf ein Objekt ohne Name verweisen kann. Die folgende Funktion kann nicht mit normalen *string*-Instanzen aufgerufen werden:

```
void Foo(std::string&& s) {
  ...
}

// Erlaubt, ein temporäres Objekt hat keinen Namen
Foo(std::string("Otto"));

// Nicht erlaubt, die Variable hat einen Namen
std::string author("Josuttis");
Foo(author);
```

Da namenlose Objekte weiter nicht mehr benutzt werden können, ist es zulässig, das Objekt durch den Funktionsaufruf kaputt zu machen. Das können Sie ausnutzen, um effizientere Funktionen zu schreiben.

Was steckt dahinter?

Rvalues sind im Gegensatz zu *Lvalues* Objekte, die keine Adresse im Speicher haben und deshalb auch keinen Namen. Dadurch können sie nur an der Definitionsstelle benutzt werden und sind abgesehen davon nicht zugänglich. Die Bezeichnungen *Rvalue* und *Lvalue* kommen ursprüng-