O'REILLY®

Effektives modernes C++

42 TECHNIKEN FÜR BESSEREN C++11- UND C++14-CODE

Scott Meyers
Übersetzung von Thomas Demmig

Effektives modernes C++

Scott Meyers

Deutsche Übersetzung
von Thomas Demmig

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

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

Die Originalausgabe erschien 2014 unter dem Titel *Effective Modern C++* bei O'Reilly Media, Inc.

Die Darstellung einer Königsfruchttaube im Zusammenhang mit dem Thema C++ ist ein Warenzeichen des O'Reilly Verlags GmbH & Co. KG

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

Lektorat: Alexandra Follenius, Köln
Übersetzung: Thomas Demmig, Mannheim
Korrektorat: Friederike Daenecke, Zülpich
Umschlaggestaltung: Michael Oreal, Köln
Produktion: Andrea Miß, Köln
Satz: Reemers Publishing Services GmbH, Krefeld,
www.reemers.de,
Belichtung, Druck und buchbinderische Verarbeitung:
Mediaprint, Paderborn

ISBN: 978-3-95875-049-4

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Für Darla,
einen außergewöhnlichen schwarzen Labrador-Retriever

Danksagung

Ich begann mit meinen Nachforschungen rund um C++0x (das dann zu C++11 werden sollte) im Jahr 2009. In der Usenet-Newsgroup `comp.std.c++` stellte ich viele Fragen, und ich bin den Mitgliedern dieser Community (insbesondere Daniel Krügler) für ihre sehr hilfreichen Postings ausgesprochen dankbar. In den letzten Jahren habe ich mich dann eher auf Stack Overflow[1] herumgetrieben, wenn ich Fragen zu C++11 und C++14 hatte, und auch da bin ich der Community genauso für ihre Hilfe dankbar, die mich bei den Details des modernen C++ nicht allein gelassen hat.

2010 habe ich Materialien für einen Kurs zu C++0x vorbereitet (der schließlich als *Overview of the New C++*[2] von Artima Publishing im Jahr 2010 veröffentlicht wurde). Sowohl diese Materialien als auch mein Wissen profitierten von den Testlesern Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober und Anthony Williams. Ohne ihre Hilfe hätte ich mich bestimmt nicht an *Effektives Modernes C++* herangewagt. Der englische Titel *Effective Modern C++* wurde übrigens von einer Reihe von Lesern vorgeschlagen, als ich am 18. Februar 2014 mein Blog-Posting »Help me name my book«[3] veröffentlichte und Andrei Alexandrescu (Autor von *Modern C++ Design*[4], Addison-Wesley, 2001) war so freundlich, dem Titel seinen Segen zu geben und nicht darauf zu bestehen, dass dies sein Begriff sei.

Ich kann nicht mehr alle Quellen angeben, auf denen die Informationen in diesem Buch beruhen, aber manche hatten dann doch einen ziemlich direkten Einfluss. Die Verwendung eines nicht definierten Templates in Technik 4, um dem Compiler Typ-Informationen zu entlocken, wurde von Stephan T. Lavavej vorgeschlagen, während mich Matt P. Dziubinski auf `Boost.TypeIndex` aufmerksam machte. In Technik 5 stammt das Beispiel mit dem `unsigned-std::vector<int>::size_type` aus Andrey Karpovs Artikel »In what way can C++0x standard help you eliminate 64-bit errors«[5] vom 28. Februar 2010. Das Beispiel rund um `std::pair<std::string, int>/std::pair<const std::string int>` aus der

1 http://stackoverflow.com/
2 http://www.artima.com/shop/overview_of_the_new_cpp
3 http://scottmeyers.blogspot.com/2014/02/help-me-name-my-book.html
4 http://erdani.com/index.php/books/modern-c-design/
5 http://www.viva64.com/en/b/0060/

gleichen Technik stammt aus Stephan T. Lavavejs Vortrag »STL11: Magic && Secrets«[6], den er auf der *Going Native 2012* gehalten hat. Technik 6 wurde von Herb Sutters Artikel »GotW #94 Solution: AAA Style (Almost Always Auto)«[7] vom 12. August 2013 inspiriert. Die Idee zu Technik 9 stammt von Martinho Fernandes' Blog-Post »Handling dependent names«[8] vom 27. Mai 2012. Das Beispiel aus Technik 12 mit dem Überladen von Referenz-Qualifiern basiert auf Caseys Antwort auf die Frage »Wozu kann man Member-Funktionen von Referenz-Qualifiern überladen?«[9], die am 14. Januar 2014 auf Stack Overflow gestellt wurde. Meine Behandlung der in C++14 erweiterten Unterstützung von `constexpr`-Funktionen in Technik 15 greift auf Informationen zurück, die ich von Rein Halbersma erhielt. Technik 16 basiert auf Herb Sutters Präsentation »You don't know `const` and `mutable`« von der *C++ and Beyond 2012*. Der Vorschlag in Technik 18, Fabrikfunktionen einen `std::unique_ptr` zurückgeben zu lassen, baut auf Herb Sutters Artikel »GotW# 90 Solution: Factories«[10] vom 30. Mai 2013 auf. In Technik 19 ist `fastLoadWidget` von Herb Sutters Präsentation »My Favorite C++ 10-Liner«[11] auf der *Going Native 2013* inspiriert. Meine Behandlung von `std::unique_ptr` und unvollständigen Typen in Technik 22 nutzt Herb Sutters Artikel »GotW #100: Compilation Firewalls«[12] vom 27. November 2011, aber auch Howard Hinnants Antwort vom 22. Mai 2011 auf die Stack-Overflow-Frage »Muss std::unique_ptr<T> die vollständige Definition von T kennen?«[13]. Das `Matrix`-Additions-Beispiel aus Technik 25 basiert auf Texten von David Abrahams. JoeArgonnes Kommentar vom 8. Dezember 2012 zum Blog-Post »Another alternative to lambda move capture«[14] vom 30. November 2012 diente als Quelle für das `std::bind`-basierte Vorgehen in Technik 32, um Init Captures in C++11 zu emulieren. Die Erläuterungen in Technik 37 zum Problem mit einem impliziten Detach im Destruktor von `std::thread` stammen aus Hans-J. Boehms Artikel »N2802: A plea to reconsider detach-on-destruction for thread objects«[15] vom 4. Dezember 2008. Technik 41 wurde ursprünglich durch eine Diskussion in den Kommentaren zu David Abrahams Blog-Post »Want speed? Pass by value.«[16] vom 15. August 2009 inspiriert. Die Idee, dass Move-Only-Typen eine besondere Behandlung benötigen, stammt von Matthew Fioravante, während die Analyse des zuweisungs-basierten Kopierens auf Kommentaren von Howard Hinnant aufbaut. In Technik 42 haben mir Stephan T. Lavavej und Howard Hinnant dabei geholfen, die relativen Performance-Profile von Emplacement- und Insertion-Funktionen zu verstehen, während mich Michael Winterberg darauf aufmerksam machte, wie ein Emplacement zu Ressourcenlecks führen kann. (Michael bezieht sich dabei auf Sean

6 http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/STL11-Magic-Secrets
7 http://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/
8 http://flamingdangerzone.com/cxx11/2012/05/27/dependent-names-bliss.html
9 http://stackoverflow.com/questions/21052377/whats-a-use-case-for-overloading-member-functions-on-reference-qualifiers
10 http://herbsutter.com/2013/05/30/gotw-90-solution-factories/
11 http://channel9.msdn.com/Events/GoingNative/2013/My-Favorite-Cpp-10-Liner
12 http://herbsutter.com/gotw/_100/
13 http://stackoverflow.com/questions/6012157/is-stdunique-ptrt-required-to-know-the-full-definition-of-t
14 http://jrb-programming.blogspot.com/2012/11/another-alternative-to-lambda-move.html
15 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2802.html
16 http://web.archive.org/web/20140113221447/http:/cpp-next.com/archive/2009/08/want-speed-pass-by-value/

Parents Präsentation »C++ Seasoning«[17] von der *Going Native 2013*.) Auch hat er beschrieben, wie Emplacement-Funktionen die direkte Initialisierung nutzen, während Insertion-Funktionen eine Copy-Initialisierung verwenden.

Das Korrekturlesen von Entwürfen eines technisch orientierten Buchs ist eine anspruchsvolle, zeitaufwendige und kritische Aufgabe, und ich bin froh, dass so viele Leute das für mich getan haben. Komplette Versionen oder Teile von *Effective Modern C++* wurden von Cassio Neri, Nate Kohl, Gerhard Kreuzer, Leor Zolman, Bart Vandewoestyne, Stephan T. Lavavej, Nevin »:-)« Liber, Rachel Cheng, Rob Stewart, Bob Steagall, Damien Watkins, Bradley E. Needham, Rainer Grimm, Fredrik Winkler, Jonathan Wakely, Herb Sutter, Andrei Alexandrescu, Eric Niebler, Thomas Becker, Roger Orr, Anthony Williams, Michael Winterberg, Benjamin Huchley, Tom Kirby-Green, Alexey A Nikitin, William Dealtry, Hubert Matthews und Tomasz Kamiński offiziell Korrektur gelesen. Zudem erhielt ich Kommentare von vielen Lesern über O'Reilly's Early Release EBooks[18] und Safari Books Online's Rough Cuts[19], über Kommentare in meinem Blog (*The View from Aristeia*[20]) und per E-Mail. Ich bedanke mich bei jedem für seine Hilfe, denn so konnte das Buch *viel* besser werden. Ich bin insbesondere Stephan T. Lavavej und Rob Stewart zu großem Dank verpflichtet, da mich ihre außerordentlich detaillierten und umfassenden Anmerkungen vermuten lassen, dass sie genauso viel Zeit mit diesem Buch verbracht haben wie ich. Ein besonderer Dank geht zudem an Leor Zolman, der neben dem Lesen des Skripts auch noch alle Codebeispiele doppelt kontrollierte.

Speziell die digitalen Versionen dieses Buchs wurden zur Kontrolle von Gerhard Kreuzer, Emyr Williams und Bradley E. Needham gelesen.

Meine Entscheidung, die Zeilenlänge in den (englischsprachigen) Codebeispielen auf 64 Zeichen zu beschränken (wodurch sich der Code sowohl im Druck als auch auf vielen digitalen Geräten unabhängig von Ausrichtung und Schriftart ordentlich anzeigen lässt), basiert auf Daten von Michael Maher.

Ashley Morgan Williams sorgte dafür, dass das Essen im Lake Oswego Pizzicato immer besonders unterhaltsam war. Wenn es um gigantischen Caesar-Salat geht, ist sie die richtige Adresse.

Über 20 Jahre nach meinem ersten Einsatz als Autor hat meine Frau Nancy L. Urbano erneut viele Monate eingeschränkter Kommunikation mit mir mit einer Mischung aus Resignation, Verzweiflung sowie Verständnis und Unterstützung im richtigen Moment ertragen. In der gleichen Zeit hat sich unser Hund Darla einen Großteil der Zeit damit zufriedengegeben, stundenlang zu dösen, während ich auf den Computermonitor starrte. Aber Darla vergaß nie, mich daran zu erinnern, dass es ein Leben jenseits der Tastatur gibt.

17 http://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasonin
18 http://shop.oreilly.com/category/early-release.do
19 http://my.safaribooksonline.com/roughcuts
20 http://scottmeyers.blogspot.com/

Einleitung

Wenn Sie ein erfahrener C++-Entwickler und ein bisschen so wie ich sind, haben Sie beim Erscheinen von C++11 gedacht: »Ja, ja, ich hab' schon verstanden. Wie C++, nur ein bisschen mehr.« Aber als Sie sich näher damit befasst haben, waren Sie überrascht, was die Änderungen bewirkten. `auto`-Deklarationen, Range-basierte `for`-Schleifen, Lambda-Ausdrücke und Rvalue-Referenzen haben C++ geändert – ganz abgesehen von den neuen Concurrency-Features. Und dann gibt es noch die sprachlichen Änderungen. `0` und `typedef` sind out, `nullptr` und Alias-Deklarationen sind in. Enums sollten jetzt einen Gültigkeitsbereich haben. Smart Pointer sind den eingebauten vorzuziehen. Das Verschieben von Objekten ist im Allgemeinen besser als das Kopieren.

Es gibt viel Neues zu lernen in C++11 und in C++14.

Wichtiger ist aber noch, dass es viel Neues zu lernen gibt in Bezug auf den *effektiven* Einsatz dieser neuen Möglichkeiten. Wenn Sie grundlegende Informationen über »moderne« C++-Features suchen, finden Sie allerorten etwas. Aber wollen Sie erfahren, wie Sie diese Features einsetzen, um korrekte, effiziente, wartbare und portable Software zu schreiben, wird es schon schwieriger. Hier kommt dieses Buch ins Spiel. Es soll nicht nur die Features von C++11 und C++14 beschreiben, sondern auch ihren effektiven Einsatz erklären.

Die Informationen in diesem Buch sind in *Techniken* unterteilt. Sie wollen die verschiedenen Formen der Typableitung verstehen? Oder wissen, wann Sie `auto`-Deklarationen einsetzen (und wann nicht)? Interessiert es Sie, zu erfahren, warum `const`-Memberfunktionen Thread-sicher sein sollen, wie Sie das Pimpl-Idiom mithilfe von `std::unique_ptr` implementieren, warum Sie Default-Capture-Modi in Lambda-Ausdrücken vermeiden sollten oder was die Unterschiede zwischen `std::atomic` und `volatile` sind? Die Antworten finden Sie alle in diesem Buch. Zudem sind sie plattformunabhängig und standardkonform. In diesem Buch geht es um *portables* C++.

Die Techniken in diesem Buch sind Empfehlungen, keine Gesetze, denn Empfehlungen haben Ausnahmen. Der wichtigste Teil jeder Technik ist nicht der eigentliche Ratschlag, sondern die Gründe, die dahinter stehen. Haben Sie diese gelesen, können Sie selbstständig erkennen, ob die Umstände in Ihrem Projekt ein Ignorieren der Empfehlungen

einer Technik rechtfertigen. Das Buch soll Ihnen nicht einfach nur sagen, was Sie tun oder lassen sollten, sondern es soll Ihnen ein tiefer gehendes Verständnis davon vermitteln, wie C++11 und C++14 funktionieren.

Begriffe und Konventionen

Um sicherzustellen, dass wir uns verstehen, müssen wir uns auf ein paar Begriffe einigen. Ironischerweise beginnt das mit »C++«. Es gab bisher vier offizielle Versionen von C++, die jeweils nach dem Jahr benannt wurden, in dem der entsprechende ISO-Standard veröffentlicht wurde: *C++98*, *C++03*, *C++11* und *C++14*. C++98 und C++03 unterscheiden sich nur in technischen Details, daher fasse ich beide in diesem Buch unter der Bezeichnung »C++98« zusammen. Schreibe ich über C++11, meine ich sowohl C++11 als auch C++14, da es sich bei C++14 letztendlich um eine Obermenge von C++11 handelt. Rede ich von C++14, meine ich auch speziell C++14. Und schreibe ich einfach von »C++«, beziehen sich die Erläuterungen auf alle Sprachversionen.

Verwendeter Begriff	Sprachversion
C++	Alle
C++98	C++98 und C++03
C++11	C++11 und C++14
C++14	C++14

So schreibe ich vielleicht, dass C++ vor allem Wert auf Effizienz legt (das gilt für alle Versionen), dass C++98 keine Unterstützung für Concurrency besitzt (gilt nur für C++98 und C++03), dass C++11 Lambda-Ausdrücke unterstützt (gilt für C++11 und C++14) und dass C++14 eine generalisierte Ermittlung des Rückgabewerts von Funktionen bietet (gilt nur für C++14).

Das wahrscheinlich allgegenwärtigste Feature von C++11 ist die Move-Semantik – und deren Grundlage ist das Unterscheiden von Ausdrücken in *Rvalues* und *Lvalues*. Denn Rvalues zeichnen Objekte als für Move-Operationen geeignet aus, während das bei Lvalues im Allgemeinen nicht der Fall ist. Laut Konzept (allerdings nicht immer in der Praxis) entsprechen Rvalues temporären Objekten, die von Funktionen zurückgegeben werden, während Lvalues Objekte sind, die Sie referenzieren können – entweder per Name oder über einen Zeiger oder eine Lvalue-Referenz.

Um zu bestimmen, ob ein Ausdruck ein Lvalue ist, ist es hilfreich, sich zu fragen, ob Sie seine Adresse erhalten können. Ist das der Fall, handelt es sich im Allgemeinen um einen Lvalue. Wenn nicht, ist es meist ein Rvalue. Eine nette Eigenschaft dieser Heuristik: Sie denken daran, dass der Typ eines Ausdrucks unabhängig davon ist, ob es sich beim Ausdruck um einen Lvalue oder einen Rvalue handelt. Ist also ein Typ `T` gegeben, können Sie Lvalues vom Typ `T`, aber auch Rvalues vom Typ `T` haben. Das ist besonders dann wichtig, wenn Sie mit einem Parameter eines Rvalue-Referenz-Typs arbeiten, denn der Parameter selbst ist dann ein Lvalue:

```
class Widget {
public:
  Widget(Widget&& rhs);    // rhs ist ein Lvalue, obwohl es einen
  …                        // Rvalue-Referenz-Typ besitzt
};
```

Hier ist es in Ordnung, die Adresse von `rhs` im Inneren des Move-Konstruktors von `Widget` einzusetzen, daher handelt es sich bei `rhs` um einen Lvalue, obwohl dessen Typ eine Rvalue-Referenz ist. (Aus ähnlichen Gründen sind alle Parameter Lvalues.)

Dieser Codeausschnitt demonstriert eine Reihe von Konventionen, denen ich im Allgemeinen folge:

- Der Klassenname ist `Widget`. Ich nutze `Widget` immer dann, wenn ich mich auf einen beliebigen benutzerdefinierten Typ beziehen will. Wenn ich nicht gerade spezifische Details der Klasse zeigen will, greife ich auf `Widget` zu, ohne es zu deklarieren.
- Ich verwende den Parameternamen *rhs* (»Right-Hand Side«). Das ist mein bevorzugter Name für die *Move-Operationen* (also den Move-Konstruktor und den Move-Zuweisungsoperator) und die *Copy-Operationen* (also den Copy-Konstruktor und den Copy-Zuweisungsoperator). Ich nutze ihn auch für den rechtsseitigen Parameter von binären Operatoren:

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

  Es sollte Sie (hoffentlich) nicht überraschen, dass ich *lhs* für »Left-Hand Side« verwende.
- Bestimmte Teile des Codes oder von Kommentaren sind besonders hervorgehoben, um Ihre Aufmerksamkeit darauf zu lenken. Im oben gezeigten Move-Konstruktor von `Widget` habe ich die Deklaration von `rhs` und den Teil des Kommentars hervorgehoben, der erwähnt, dass es sich bei `rhs` um einen Lvalue handelt. Hervorgehobener Code ist nicht automatisch besonders gut oder schlecht. Es handelt sich schlicht um Code, den Sie sich genauer anschauen sollten.
- Ich verwende »…«, um zu zeigen, dass an dieser Stelle anderer Code eingefügt werden kann. Diese enge Ellipse unterscheidet sich von der weiten Ellipse (»...«), die im Quellcode für die Variadic Templates von C++11 verwendet wird. Das klingt verwirrend, ist es aber nicht. Zum Beispiel:

  ```
  template<typename... Ts>               // Das sind C++-
  void processVals(const Ts&... params)  // Ellipsen (echter
  {                                      // Quellcode).

    \u                                    // Das heißt: "hier
                                         // Code einfügen"
  }
  ```

  Die Deklaration von `processVals` zeigt, dass ich beim Deklarieren von Typ-Parametern in Templates `typename` verwende. Das ist aber eine persönliche Vorliebe – das Schlüsselwort `class` würde genauso funktionieren. Zeige ich Codeausschnitte aus einem C++-Standard, deklariere ich Typ-Parameter mittels `class`, denn das wird im Standard so gemacht.

Wird ein Objekt mithilfe eines anderen Objekts des gleichen Typs initialisiert, wird das neue Objekt als *Kopie* des initialisierenden Objekts bezeichnet, auch wenn diese Kopie mittels des Move-Konstruktors erstellt wurde. Leider gibt es keine Terminologie in C++, die zwischen einem Objekt einer per Copy und einer per Move erzeugten Kopie unterscheidet:

```
void someFunc(Widget w);        // Parameter w
                                // wird by Value übergeben

Widget wid;                     // wid ist ein Widget

someFunc(wid);                  // In diesem Aufruf ist
                                // w eine Kopie von wid, die per
                                // Copy-Erzeugung
                                // erstellt wird.

someFunc(std::move(wid));       // In diesem Aufruf ist
                                // w eine Kopie von wid, die per
                                // Move-Erzeugung
                                // erstellt wird.
```

Kopien von Rvalues sind im Allgemeinen Move-erzeugt, während Kopien von Lvalues Copy-erzeugt werden. Eine Folge davon ist, dass Sie nicht sagen können, wie teuer das Erstellen einer Kopie war, wenn Sie nur wissen, dass es sich um eine Kopie handelt. Im obigen Code kann man zum Beispiel nicht sagen, wie teuer es ist, den Parameter `w` zu erstellen, ohne zu wissen, ob Rvalues oder Lvalues an `someFunc` übergeben werden. (Sie müssten zudem noch die Kosten für das Verschieben und Kopieren von `Widgets` kennen.)

In einem Funktionsaufruf sind die übergebenen Ausdrücke die *Argumente* der Funktion. Sie werden genutzt, um die *Parameter* der Funktion zu initialisieren. Im ersten Aufruf von `someFunc` im obigen Code ist das Argument `wid`. Im zweiten Aufruf ist es `std::move(wid)`. In beiden Fällen ist der Parameter `w`. Es ist wichtig, zwischen Argumenten und Parametern zu unterscheiden, denn Parameter sind Lvalues, während Argumente abhängig von der Art der Initialisierung Rvalues oder Lvalues sein können. Das ist besonders während des Prozesses des *Perfect Forwarding* wichtig, bei dem ein an eine Funktion übergebenes Argument so an eine zweite Funktion weitergereicht wird, dass die »Rvalueness« oder »Lvalueness« erhalten bleibt. (Perfect Forwarding wird im Detail in Technik 30 besprochen.)

Sauber entworfene Funktionen sind *Exception-sicher*, sie bieten also grundlegende Garantien zur Exception-Sicherheit (die *Basic Guarantee*). Solche Funktionen garantieren dem Aufrufer, dass selbst in dem Fall, dass eine Ausnahme ausgelöst wird, die Programm-Invarianten intakt bleiben (also keine Datenstrukturen zerstört werden) und keine Ressourcenlecks entstehen. Funktionen, die eine starke Ausnahme-Sicherheit garantieren (die *Strong Guarantee*), stellen für den Aufrufer sicher, dass der Status des Programms bei einer Ausnahme so bleibt, wie er vor dem Aufruf war.

Rede ich von einem *Funktionsobjekt*, meine ich im Allgemeinen ein Objekt eines Typs, der eine Memberfunktion `operator()` anbietet. Mit anderen Worten: ein Objekt, das sich wie eine Funktion verhält. Gelegentlich nutze ich den Begriff etwas allgemeiner und meine

dann alles, was mit der Syntax einer Nicht-Memberfunktion aufgerufen werden kann (also »*functionName(arguments)*«). Diese umfassendere Definition enthält nicht nur Objekte, die `operator()` anbieten, sondern auch Funktionen und C-Funktionszeiger. (Die engere Definition stammt aus C++98, die weitere aus C++11.) Verallgemeinern wir noch weiter, indem wir Zeiger auf Memberfunktionen hinzugesellen, kommen wir zu den *aufrufbaren Objekten* (Callable Objects). Sie können die feinen Unterschiede normalerweise ignorieren und sich Funktionsobjekte sowie aufrufbare Objekte als Elemente in C++ vorstellen, die wie eine Funktion aufgerufen werden können.

Funktionsobjekte, die per Lambda-Ausdruck erstellt wurden, nennt man auch *Closures*. Man muss nur selten zwischen Lambda-Ausdrücken und den durch sie erzeugten Closures unterscheiden, daher schreibe ich häufig über beide als *Lambdas*. Ebenso mache ich nur selten einen Unterschied zwischen *Funktions-Templates* (also Templates, die Funktionen erzeugen) und *Template-Funktionen* (also den Funktionen, die aus Funktions-Templates erzeugt wurden). Dasselbe gilt für *Klassen-Templates* und *Template-Klassen*.

Sie können in C++ vieles sowohl deklarieren als auch definieren. Durch *Deklarationen* werden Namen und Typen eingeführt, ohne weitere Details zu liefern – zum Beispiel, wo Speicher genutzt oder Dinge implementiert werden sollen:

```
extern int x;                       // Objektdeklaration

class Widget;                       // Klassendeklaration

bool func(const Widget& w);         // Funktionsdeklaration

enum class Color;                   // Scoped Enum-Deklaration
                                    // (siehe Technik 10)
```

Definitionen legen die Speicherorte oder Implementierungsdetails fest:

```
int x;                              // Objektdefinition

class Widget {                      // Klassendefinition
  …
};

bool func(const Widget& w)
{ return w.size() < 10; }           // Funktionsdefinition

enum class Color
{ Yellow, Red, Blue };              // Scoped Enum-Definition
```

Eine Definition ist gleichzeitig auch eine Deklaration. Sofern es also nicht ausgesprochen wichtig ist, dass es sich bei etwas um eine Definition handelt, tendiere ich eher dazu, von Deklarationen zu schreiben.

Ich definiere die *Signatur* einer Funktion als den Teil ihrer Deklaration, der Parameter- und Rückgabetypen festlegt. Funktions- und Parameternamen sind nicht Teil der Signatur. Im obigen Beispiel hat `func` die Signatur `bool(const Widget&)`. Andere Elemente einer Funktionsdeklaration neben den Parameter- und Rückgabetypen (zum Beispiel `noexcept` oder `constexpr`, sofern vorhanden) gehören auch nicht dazu. (`noexcept` und `constexpr` werden in

den Techniken 14 und 15 beschrieben.) Die offizielle Definition einer »Signatur« unterscheidet sich von meiner ein wenig, aber für dieses Buch ist meine Version nützlicher. (In der offiziellen Definition gehören Rückgabetypen manchmal nicht dazu.)

Neue C++-Standards sind im Allgemeinen abwärtskompatibel, sodass sich auch älterer Code weiter übersetzen lässt, aber manchmal setzt das Standardization Committee Features auf *deprecated* (veraltet, abgekündigt). Solche Features stehen auf der Abschussliste des Komitees und werden eventuell in zukünftigen Standards entfernt. Compiler können Warnungen beim Einsatz von deprecated Features ausgeben, müssen es aber nicht. Trotzdem ist es am besten, sie ganz zu vermeiden. Denn es kann nicht nur in Zukunft zu Portierungsaufwänden führen, sondern die Features sind im Allgemeinen auch schlechter als ihr neuer Ersatz. So ist zum Beispiel `std::auto_ptr` in C++11 deprecated, da `std::unique_ptr` die Aufgabe übernommen hat – und dies auch noch besser macht.

Manchmal steht im Standard, dass das Ergebnis einer Operation ein *undefiniertes Verhalten* ist. Das heißt, das Verhalten zur Laufzeit ist nicht vorhersagbar, und Sie sollten derartige Unsicherheiten unbedingt vermeiden. Beispiele von Aktionen mit undefiniertem Verhalten sind der Einsatz von eckigen Klammern (»`[]`«), um den Index über die Grenzen eines `std::vector` hinaus einzusetzen, das Dereferenzieren eines nicht initialisierten Iterators oder die Teilnahme an einem Data Race (das ist eine Situation, in der es zwei oder mehr Threads gibt, von denen mindestens einer schreibend ist und, die gleichzeitig auf den gleichen Speicherort zugreifen).

Die eingebauten Zeiger, die zum Beispiel von `new` zurückgegeben werden, nenne ich *Raw Pointer*. Das Gegenstück sind *Smart Pointer*. Sie überladen meist die Operatoren zum Dereferenzieren von Zeigern (`operator->` und `operator*`), wobei Technik 20 beschreibt, dass `std::weak_ptr` da eine Ausnahme ist.

In Quellcode-Kommentaren kürze ich »Konstruktor« manchmal als *`Ctor`* und »Destruktor« als *`Dtor`* ab.

Fehler und Verbesserungsvorschläge

Ich habe versucht, dieses Buch mit klaren, genauen und nützlichen Informationen zu füllen, aber es gibt sicher Wege, dies noch besser zu machen. Wenn Sie Fehler jeglicher Art finden (technische, grammatikalische, typografische, falsche Erklärungen und so weiter) oder wenn Sie Vorschläge haben, wie das Buch verbessert werden könnte, schreiben Sie mir bitte an *emc++@aristeia.com*. In Folgeauflagen habe ich die Möglichkeit, *Effektives Modernes C++* zu überarbeiten, aber ich kann nur Dinge angehen, von denen ich weiß!

Die Liste der Dinge, von denen ich schon weiß, finden Sie auf der Errata-Seite des Buches *http://www.aristeia.com/BookErrata/emc++-errata.html*.

Verwendung der Codebeispiele

Dieses Buch ist dazu gedacht, Ihnen bei der Erledigung Ihrer Arbeit zu helfen. Im Allgemeinen dürfen Sie den Code in diesem Buch in Ihren eigenen Programmen oder Dokumentationen verwenden. Solange Sie den Code nicht in großem Umfang reproduzieren, brauchen Sie uns nicht um Erlaubnis zu bitten. Zum Beispiel benötigen Sie nicht unsere Erlaubnis, wenn Sie ein Programm unter Zuhilfenahme mehrerer Codestücke aus diesem Buch schreiben. Wenn Sie allerdings einen Datenträger mit Beispielen aus O'Reilly-Büchern verkaufen oder vertreiben wollten, müssen Sie eine Genehmigung von uns einholen. Eine Frage mit einem Zitat oder einem Codebeispiel aus dem Buch zu beantworten, erfordert keine Genehmigung. Signifikante Teile von Beispielcode aus dem Buch für die eigene Produktdokumentation zu verwerten, ist dagegen genehmigungspflichtig. Wir freuen uns über eine Quellenangabe, verlangen sie aber nicht unbedingt. Zu einer Quellenangabe gehören normalerweise Autor, Titel, Verlagsangabe, Veröffentlichungsjahr und ISBN, hier also: »Scott Meyers, Effektives modernes C++, O'Reilly Verlag 2015, ISBN 978-3-95875-049-4«.

Sollten Sie das Gefühl haben, Ihre Verwendung der Codebeispiele könnte gegen das Fairnessprinzip oder die Genehmigungspflicht verstoßen, dann nehmen Sie bitte unter *kommentar@oreilly.de* Kontakt mit uns auf.

Kontakt

Bitte richten Sie Anfragen und Kommentare zu diesem Buch an den Verlag:

O'Reilly Verlag GmbH & Co. KG
Balthasarstraße 81
50670 Köln

Wir haben eine Webseite zu diesem Buch eingerichtet, auf der Errata, die Codebeispiele und zusätzliche Informationen veröffentlicht werden. Sie finden die Seite unter:

http://www.oreilly.de/catalog/effectivemodcplusger/

Kommentare oder technische Fragen zu diesem Buch schicken Sie bitte per E-Mail an:

kommentar@oreilly.de

Weitere Informationen zum gesamten Angebot des O'Reilly Verlags finden Sie auf unserer Website: *http://www.oreilly.de*.

Wir sind auf Facebook: facebook.com/oreilly.de (*https://www.facebook.com/oreilly.de*)

Folgen Sie uns auf Twitter: twitter.com/OReilly_Verlag/(*https://twitter.com/oreilly-verlag*)

KAPITEL 1

Typen ableiten

In C++98 gab es genau einen Regelsatz für die Typableitung: den für Funktions-Templates. C++11 passt diesen Regelsatz ein wenig an und fügt zwei weitere hinzu – einen für `auto` und einen für `decltype`. C++14 erweitert dann die Anwendungsbereiche für `auto` und `decltype`. Die immer weiter gehende automatische Typableitung befreit Sie von der Tyrannei, Typen hinschreiben zu müssen, die offensichtlich oder redundant sind. C++-Software lässt sich dadurch besser anpassen, da das Ändern eines Typs an einer Stelle im Quellcode durch die Typableitung automatisch dafür sorgt, dass dies auch an anderen Stellen wirksam wird. Allerdings kann es auch schwieriger werden, Code zu analysieren, da die von den Compilern ermittelten Typen nicht immer so offensichtlich sind, wie Sie es sich vielleicht erhoffen.

Ohne ein solides Verständnis der Typableitung ist effektives Programmieren in modernem C++ so gut wie unmöglich. Es gibt einfach zu viele Gelegenheiten, bei denen Typableitung geschieht: in Aufrufen von Funktions-Templates, in den meisten Situationen mit `auto`, in `decltype`-Ausdrücken und – mit C++14 – beim Einsatz des mysteriösen `decltype(auto)`.

In diesem Kapitel finden Sie die Informationen zur Typableitung, die jeder C++-Entwickler kennen muss. Es beschreibt, wie die Typableitung bei Templates funktioniert, wie `auto` darauf aufbaut und wie `decltype` seinen eigenen Weg geht. Zudem wird sogar erklärt, wie Sie Ihren Compiler dazu zwingen, die Ergebnisse seiner Typableitungen anzuzeigen, sodass Sie prüfen können, ob er so vorgeht, wie Sie es sich vorgestellt haben.

Technik 1: Typableitung beim Template

Wenn die Anwender eines komplexen Systems sich nicht darum scheren, wie es funktioniert – solange sie mit dem Ergebnis zufrieden sind – sagt das viel über das Design des Systems aus. Daran gemessen ist die Template-Typableitung in C++ ausgesprochen erfolgreich. Millionen von Programmierern haben Argumente erfolgreich an Template-Funktionen übergeben, obwohl die meisten von Ihnen höchstens in sehr groben Zügen erklären könnten, wie die von diesen Funktionen genutzten Typen ermittelt werden.

Wenn Sie sich auch zu dieser Gruppe zählen, habe ich eine gute und eine schlechte Nachricht für Sie. Die gute Nachricht ist, dass die Typableitung für Templates die Grundlage für eines der überzeugendsten Features in modernem C++ ist: `auto`. Waren Sie bisher zufrieden damit, wie in C++ Typen für Templates ermittelt wurden, werden Sie auch mit der Typableitung via `auto` in C++11 glücklich werden. Die schlechte Nachricht ist, dass beim Anwenden der Regeln zur Template-Typableitung auf `auto` manchmal weniger intuitive Ergebnisse herauskommen. Aus diesem Grund ist es wichtig, die Aspekte der Template-Typableitung wirklich zu verstehen, auf die `auto` baut. In dieser Technik werden Ihnen die notwendigen Informationen dazu vermittelt.

Wenn Sie nichts gegen ein bisschen Pseudocode haben, können wir uns ein Funktions-Template wie folgt vorstellen:

```
template<typename T>
void f(ParamType param);
```

Ein Aufruf kann dann so aussehen:

```
f(expr);                    // f mit einem Ausdruck aufrufen
```

Während des Kompilierens nutzt der Compiler *expr*, um zwei Typen abzuleiten: einen für `T` und einen für *ParamType*. Diese Typen sind meist unterschiedlich, weil *ParamType* häufig noch Ausschmückungen wie `const` oder Referenz-Qualifier enthält. Ist das Template zum Beispiel so deklariert:

```
template<typename T>
void f(const T& param);     // ParamType ist const T&.
```

und haben wir diesen Aufruf:

```
int x = 0;

f(x);                       // Aufruf mit int
```

dann wird `T` als `int` ermittelt, während *ParamType* ein `const int&` ist.

Es ist nur natürlich, davon auszugehen, dass der für `T` ermittelte Typ der gleiche ist wie der Typ des an die Funktion übergebenen Arguments – also dass `T` dem Typ von *expr* entspricht. Im obigen Beispiel ist das auch der Fall: `x` ist ein `int`, und `T` wird als `int` abgeleitet. Aber das funktioniert nicht immer so. Der Typ, der für `T` ermittelt wird, hängt nicht nur vom Typ von *expr* ab, sondern auch noch von der Form von *ParamType*. Es gibt drei Fälle:

- *ParamType* ist ein Zeiger- oder Referenztyp, aber keine universelle Referenz. (Universelle Referenzen werden in Technik 24 beschrieben. Hier müssen Sie nur wissen, dass es sie gibt und dass sie nicht dasselbe sind wie Lvalue- oder Rvalue-Referenzen.)
- *ParamType* ist eine universelle Referenz.
- *ParamType* ist weder ein Zeiger noch eine Referenz.

Wir haben also drei Szenarien bei der Typableitung, die wir uns anschauen wollen. Jedes Szenario wird dabei auf unserer allgemeinen Form für Templates aufbauen:

```
template<typename T>
void f(ParamType param);

f(expr);                // T und ParamType aus expr ableiten
```

Fall 1: ParamType ist eine Referenz oder ein Zeiger, aber keine universelle Referenz

In der einfachsten Situation ist *ParamType* ein Referenz- oder Zeigertyp, aber keine universelle Referenz. In diesem Fall funktioniert die Typableitung so:

1. Ist der Typ von *expr* eine Referenz, ignoriere den Referenz-Teil.
2. Dann vergleiche den Typ von *expr* per Mustererkennung mit *ParamType*, um `T` zu ermitteln.

Schauen wir uns zum Beispiel dieses Template an:

```
template<typename T>
void f(T& param);       // param ist eine Referenz.
```

Dazu diese Variablendeklarationen:

```
int x = 27;             // x ist ein int.
const int cx = x;       // cx ist ein const int.
const int& rx = x;      // rx ist eine Referenz auf x als const int.
```

Die abgeleiteten Typen für `param` und `T` sind dann wie folgt:

```
f(x);                   // T ist int, param ist int&.

f(cx);                  // T ist const int,
                        // param ist const int&.

f(rx);                  // T ist const int,
                        // param ist const int&.
```

Beachten Sie beim zweiten und dritten Aufruf, dass `cx` und `rx` `const`-Werte sind, daher wird `T` als `const int` abgeleitet und der Parametertyp wird zu `const int&`. Das ist für Aufrufer wichtig. Übergeben sie ein `const`-Objekt an einen Referenzparameter, erwarten sie, dass das Objekt unverändert bleibt, der Parameter also eine Referenz auf `const` ist. Darum ist es sicher, ein `const`-Objekt an ein Template zu übergeben, dass einen Parameter `T&` erwartet: Die »`const`heit« des Objekts wird Teil des Typs, der für `T` abgeleitet wird.

Im dritten Beispiel ist beachtenswert, dass der Typ von `rx` zwar eine Referenz ist, `T` aber trotzdem als Nicht-Referenz abgeleitet wird. Das liegt daran, dass die »Referenzheit« von `rx` beim Bestimmen des Typs ignoriert wird.

Diese Beispiele enthalten alle Lvalue-Referenzparameter, aber die Typableitung funktioniert genauso bei Rvalue-Referenzparametern. Natürlich können nur Rvalue-Argumente an Rvalue-Referenzparameter übergeben werden, aber diese Einschränkung hat nichts mit der Typableitung zu tun.

Ändern wir den Typ des Parameters von `f` von `T&` in `const T&`, ändert sich das Ergebnis ein bisschen – allerdings ohne große Überraschungen. Die `const`heit von `cx` und `rx` wird weiterhin beachtet, aber weil wir jetzt davon ausgehen, dass `param` eine Referenz auf ein `const` ist, muss `const` nicht länger als Teil von `T` abgeleitet werden:

```
template<typename T>
void f(const T& param);  // param ist nun ein Ref auf const.

int x = 27;              // wie zuvor
const int cx = x;        // wie zuvor
const int& rx = x;       // wie zuvor

f(x);                    // T ist int, param ist const int&.

f(cx);                   // T ist int, param ist const int&.

f(rx);                   // T ist int, param ist const int&.
```

Wie vorher wird die Referenzheit von `rx` während der Typableitung ignoriert.

Wäre `param` statt einer Referenz ein Zeiger (oder ein Zeiger auf `const`), würde das Ganze gleich ablaufen:

```
template<typename T>
void f(T* param);        // param ist jetzt ein Zeiger.

int x = 27;              // wie zuvor
const int *px = &x;      // px ist ein Zeiger auf x als const int.


f(&x);                   // T ist int, param ist int*.

f(px);                   // T ist const int,
                         // param ist const int*.
```

Vermutlich haben Sie zum Schluss nicht mehr genau gelesen, denn die Typableitungsregeln von C++ funktionieren für Referenz- und Zeigerparameter so selbstverständlich, dass sie in niedergeschriebener Form wirklich langweilig sind. Alles ist so offensichtlich! Aber das ist ja auch genau das, was Sie bei einer automatischen Typableitung haben wollen.

Fall 2: *ParamType* ist eine universelle Referenz

Bei Templates mit universellen Referenzparametern ist es nicht mehr ganz so offensichtlich. Solche Parameter werden wie Rvalue-Referenzen deklariert (das heißt, in einem Funktions-Template mit dem Typ-Parameter `T` wird ein Typ für eine universelle Referenz als `T&&` deklariert), aber das Verhalten ist anders, wenn Lvalue-Werte übergeben werden. Die ganze Geschichte erzähle ich in Technik 24, aber hier sind schon einmal die wichtigsten Punkte:

- Ist *expr* ein Lvalue, werden sowohl `T` als auch *ParamType* als Lvalue-Referenzen abgeleitet. Das ist doppelt unerwartet. Zum einen ist es die einzige Situation bei der Template-Typableitung, in der `T` als Referenz abgeleitet wird. Zum anderen ist *ParamType* zwar mit der Syntax für eine Rvalue-Referenz deklariert, der Typ wird aber trotzdem als Lvalue-Referenz ermittelt.
- Ist *expr* ein Rvalue, gelten die »normalen« Regeln (aus Fall 1).

Zum Beispiel:

```
template<typename T>
void f(T&& param);       // param ist jetzt eine universelle Referenz.

int x = 27;              // wie zuvor
const int cx = x;        // wie zuvor
const int& rx = x;       // wie zuvor

f(x);                    // x ist ein Lvalue, daher ist T int&,
                         // param ist auch int&.

f(cx);                   // cx ist ein Lvalue, daher ist T const int&,
                         // param ist auch const int&.

f(rx);                   // rx ist ein Lvalue, daher ist T const int&,
                         // param ist auch const int&.

f(27);                   // 27 ist ein Rvalue, daher ist T int,
                         // param ist daher int&&.
```

In Technik 24 wird ausführlich erklärt, warum diese Beispiele die beschriebenen Ergebnisse liefern. Entscheidend ist hier, dass sich die Regeln zur automatischen Typableitung für universelle Referenzparameter von denen für Lvalue- oder Rvalue-Referenzparameter unterscheiden. Insbesondere unterscheidet die Typableitung bei universellen Referenzen zwischen Lvalue- und Rvalue-Argumenten. Das passiert niemals bei nichtuniversellen Referenzen.

Fall 3: ParamType ist weder ein Zeiger noch eine Referenz

Ist *ParamType* weder ein Zeiger noch eine Referenz, arbeiten wir per Wertübergabe (Pass-by-Value):

```
template<typename T>
void f(T param);         // param wird als Wert übergeben.
```

`param` enthält dann eine Kopie des übergebenen Werts – ein ganz neues Objekt. Dadurch wird auch die Regel beeinflusst, wie `T` aus *expr* abgeleitet wird:

1. Wie zuvor gilt: Ist der Typ von *expr* eine Referenz, wird der Referenz-Teil ignoriert.
2. Ist *expr* nach dem Ignorieren der Referenzheit noch `const`, wird auch das ignoriert. Ebenso, falls *expr* `volatile` ist. (`volatile`-Objekte kommen selten vor. Sie werden meist nur beim Implementieren von Gerätetreibern eingesetzt. Details dazu finden Sie in Technik 40.)

Daher:

```
int x = 27;           // wie zuvor
const int cx = x;     // wie zuvor
const int& rx = x;    // wie zuvor

f(x);                 // T und param sind beide vom Typ int.

f(cx);                // T und param sind beide vom Typ int.

f(rx);                // T und param sind immer noch vom Typ int.
```

Beachten Sie: Obwohl cx und rx const-Werte repräsentieren, ist param nicht const. Das ist durchaus sinnvoll. param ist ein Objekt, des vollständig unabhängig von cx und rx ist – eine *Kopie* von cx oder rx. Die Tatsache, dass cx und rx nicht verändert werden können, sagt nichts darüber aus, ob dies bei param auch der Fall ist. Darum wird bei *expr* eine constheit (und auch eine volatileheit) ignoriert, wenn ein Typ für param abgeleitet wird: Nur weil *expr* nicht verändert werden kann, heißt das nicht, dass eine Kopie davon ebenso konstant bleiben muss.

Es ist wichtig, sich zu merken, dass const (und volatile) nur bei Wertübergaben ignoriert wird. Wie wir gesehen haben, wird bei Referenz- oder Zeiger-const-Parametern die constheit von *expr* bei der Typableitung bewahrt. Aber stellen Sie sich jetzt den Fall vor, in dem *expr* ein const-Zeiger auf ein const-Objekt ist und *expr* an einen By-Value-param übergeben wird:

```
template<typename T>
void f(T param);          // param wird weiterhin als Wert übergeben.

const char* const ptr =   // ptr ist ein const-Zeiger auf ein const-Objekt.
  "Spaß mit Zeigern";

f(ptr);                   // Argument vom Typ const char * const übergeben
```

Hier deklariert das const auf der rechten Seite des Sterns ptr als const: ptr kann nicht auf einen anderen Ort zeigen oder auf null gesetzt werden. (Das const links vom Stern sagt, dass das, worauf das ptr zeigt – der String – const ist und daher nicht verändert werden kann.) Wird ptr an f übergeben, werden die Bits des Zeigers nach param kopiert. Der *Zeiger selbst* (ptr) *wird also als Wert übergeben*. Entsprechend der Regeln der Typableitung für Werteparameter wird die constheit von ptr ignoriert, und der für param ermittelte Typ wird const char* sein – also ein veränderbarer Zeiger auf einen const-String. Die constheit dessen, worauf ptr zeigt, wird bei der automatischen Typableitung bewahrt, aber die constheit von ptr selbst wird beim Kopieren in den neuen Zeiger param verworfen.

Array-Argumente

Damit sind die meisten Fälle behandelt, die bei der Typableitung vorkommen können. Es gibt aber einen Spezialfall, den Sie kennen sollten. Array-Typen unterscheiden sich nämlich von Zeigertypen, auch wenn sie manchmal austauschbar scheinen. Das liegt vor allem daran, dass sich in vielen Situationen ein Array in einen Zeiger auf sein erstes

Element per *Decay* umwandeln lässt. Damit wird Code wie der im folgenden Beispiel kompilierbar:

```
const char name[] = "J. P. Briggs";  // Typ von name ist
                                     // const char[13].

const char * ptrToName = name;       // Das Array wird zu einem Zeiger.
```

Hier wird der `const char*`-Zeiger `ptrToName` mit `name` initialisiert, einem `const char[13]`. Diese Typen (`const char*` und `const char[13]`) sind nicht gleich, aber aufgrund der Array-nach-Zeiger-Decay-Regel lässt sich der Code kompilieren.

Was geschieht aber, wenn ein Array an ein Template als Werteparameter übergeben wird? Was passiert dann?

```
template<typename T>
void f(T param);      // Template mit Werteparameter

f(name);              // Welche Typen werden für T und param abgeleitet?
```

Wir beginnen mit der Beobachtung, dass es kein Array als Funktionsparameter gibt. Ja, ja, die Syntax ist erlaubt:

```
void myFunc(int param[]);
```

Aber die Array-Deklaration wird als Zeigerdeklaration behandelt. `myFunc` könnte auch so deklariert werden:

```
void myFunc(int* param);         // gleiche Funktion wie oben
```

Diese Äquivalenz von Array- und Zeigerparametern resultiert aus den C-Wurzeln von C++. Wegen ihr glauben viele, Array- und Zeigertypen seien das Gleiche.

Da Deklarationen von Array-Parametern so behandelt werden, als handele es sich um Zeigerparameter, wird der Typ eines an eine Template-Funktion als By-Value übergebenen Parameters als Zeigertyp ermittelt. Bei einem Aufruf des Templates `f` wird dessen Typparameter `T` daher als `const char*` abgeleitet:

```
f(name);          // name ist Array, aber T wird zu const char*.
```

Aber jetzt kommt die Überraschung: Funktionen können zwar keine Parameter als echte Arrays deklarieren, aber es *ist* ihnen möglich, Parameter zu deklarieren, die *Referenzen* auf Arrays sind! Wenn wir also das Template `f` so anpassen, dass es sein Argument als Referenz übernimmt,

```
template<typename T>
void f(T& param);      // Template mit Referenzparameter
```

und wir dann ein Array übergeben,

```
f(name);               // Array an f übergeben
```

ist der für `T` abgeleitete Typ tatsächlich der Typ des Arrays! Dazu gehört auch dessen Größe. Daher wird `T` in diesem Beispiel zu `const char [13]`, und der Typ des Parameters von `f` (eine Referenz auf dieses Array) ist `const char (&)[13]`. Ja, die Syntax sieht verboten

aus, aber das Wissen darüber schindet Eindruck (falls sich Ihr Gegenüber dafür interessieren sollte).

Interessanterweise können Sie durch die Fähigkeit, Referenzen auf Arrays zu deklarieren, ein Template erstellen, das die Anzahl der Elemente in einem Array ermittelt:

```
// Größe eines Arrays als beim Kompilieren konstante Größe. (Der
// Array-Parameter hat keinen Namen, weil wir uns nur für die
// Anzahl der Elemente interessieren.)

template<typename T, std::size_t N>                // siehe Info
constexpr std::size_t arraySize(T (&)[N]) noexcept  // weiter unten
{                                                  // zu constexpr
  return N;                                        // und
}                                                  // noexcept
```

Wie in Technik 15 beschrieben ist, sorgt das Deklarieren dieser Funktion als `constexpr` dafür, dass das Ergebnis schon zum Zeitpunkt des Kompilierens zur Verfügung steht. Damit lässt sich zum Beispiel ein Array mit der gleichen Anzahl an Elementen wie bei einem gegebenen Array deklarieren, dessen Größe aus einer Initialisierungsliste mit geschweiften Klammern berechnet wird:

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };      // keyVals hat
                                                 // 7 Elemente
.
int mappedVals[arraySize(keyVals)];              // genauso wie
                                                 // mappedVals
```

Natürlich bevorzugen Sie als moderner C++-Entwickler ein `std::array` gegenüber einem eingebauten Array:

```
std::array<int, arraySize(keyVals)> mappedVals;  // mappedVals
                                                 // mit Größe 7
```

Die Deklaration von `arraySize` als `noexcept` hilft dem Compiler, besseren Code zu erzeugen. Details dazu finden Sie in Technik 14.

Funktionsargumente

Arrays sind nicht die einzigen Elemente in C++, die sich in Zeiger verwandeln können. Funktionstypen können per Decay zu Funktionszeigern werden, und alles, was wir zur Typableitung für Arrays geschrieben haben, gilt auch für die Typableitung von Funktionen und ihre Umwandlung in Funktionszeiger. Als Ergebnis:

```
void someFunc(int, double);   // someFunc ist eine Funktion,
                              // Typ ist void(int, double).

template<typename T>
void f1(T param);             // In f1 wird param By-Value übergeben.

template<typename T>
void f2(T& param);            // In f2 wird param By-Ref übergeben.

f1(someFunc);                 // param als ptr-to-func bestimmt,
```