O'REILLY®

Effektives modernes C++

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

Scott Meyers
Übersetzung von Thomas Demmig

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 `const`heit (und auch eine `volatile`heit) 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 `const`heit 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 `const`heit von `ptr` ignoriert, und der für `param` ermittelte Typ wird `const char*` sein – also ein veränderbarer Zeiger auf einen `const`-String. Die `const`heit dessen, worauf `ptr` zeigt, wird bei der automatischen Typableitung bewahrt, aber die `const`heit 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,
```

```
                                // Typ ist void (*)(int, double).
f2(someFunc);                   // param als ref-to-func bestimmt,
                                // Typ ist void (&)(int, double).
```

Das macht in der Praxis nur sehr selten einen Unterschied, aber wenn Sie schon etwas über die Array-nach-Zeiger-Umwandlung wissen, sollten Sie auch über die Funktion-nach-Zeiger-Umwandlung informiert sein.

Das sind sie also – die mit `auto` in Zusammenhang stehenden Regeln für die Template-Typableitung. Ich habe am Anfang geschrieben, dass sie ziemlich einfach sind, und für den größten Teil stimmt das auch. Die Sonderbehandlung bei Lvalues beim Ermitteln von Typen für universelle Referenzen stört dieses schöne Bild ein wenig, und die Regeln für Arrays und Funktionen zum Umwandeln in Zeiger sorgen ebenfalls für Unruhe. Manchmal wollen Sie dann doch vermutlich einfach Ihren Compiler schütteln und ihn anschreien: »Sag mir, was für einen Typ du abgeleitet hast!« Wenn das geschiet, schauen Sie sich Technik 4 an, denn dort geht es darum, dem Compiler genau diese Information zu entlocken.

Was Sie sich merken sollten

- Während der Template-Typableitung werden Referenzargumente als Nicht-Referenzen behandelt – die Referenzheit wird also ignoriert.
- Beim Ableiten von Typen für universelle Referenzparameter werden Lvalue-Argumente besonders behandelt.
- Beim Ableiten von Typen für By-Value-Parameter werden `const`- und/oder `volatile`-Argumente so behandelt, als ob sie nicht mit `const` oder `volatile` versehen wären.
- Während der Template-Typableitung werden Arrays oder Funktionsnamen als Argumente in Zeiger umgewandelt, sofern sie nicht zum Initialisieren von Referenzen genutzt werden.

Technik 2: Die auto-Typableitung verstehen

Wenn Sie Technik 1 zur Template-Typableitung gelesen haben, wissen Sie schon alles, was für die Typableitung bei `auto` notwendig ist. Denn abgesehen von einer kuriosen Ausnahme sind die `auto`- und die Template-Typableitung identisch. Aber wie kann das sein? Zur Template-Typableitung gehören Templates und Funktionen und Parameter, aber bei `auto` kommt nichts davon vor.

Das stimmt, macht aber nichts. Es gibt eine direkte Abbildung zwischen der Template-Typableitung und der `auto`-Typableitung – eine algorithmische Umwandlung vom einen in das andere.

In Technik 1 wird die Template-Typableitung mit diesem Funktions-Template erklärt:

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

Dazu gehört dieser Aufruf:

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

Im Aufruf von `f` nutzt der Compiler *expr*, um die Typen für `T` und *ParamType* zu ermitteln.

Wird eine Variable mithilfe von `auto` deklariert, nimmt `auto` die Rolle von `T` im Template ein und die Typ-Spezifikation für die Variable fungiert als *ParamType*. Das lässt sich einfacher zeigen als erklären. Schauen Sie sich daher bitte dieses Beispiel an:

```
auto x = 27;
```

Hier ist die Typ-Spezifikation für `x` einfach `auto`. In der Deklaration:

```
const auto cx = x;
```

ist die Typ-Spezifikation jedoch `const auto`. Und bei:

```
const auto& rx = x;
```

ist es `const auto&`. Um Typen für `x`, `cx` und `rx` zu ermitteln, verhält sich der Compiler so, als ob es ein Template für jede Deklaration und einen Aufruf dieses Templates mit dem entsprechenden Initialisierungsausdruck gäbe:

```
template<typename T>                // konzeptionelles Template
void func_for_x(T param);           // zum Ermitteln vom Typ von x

func_for_x(27);                     // konzeptioneller Aufruf: Ermittelter
                                    // Typ von param ist Typ von x.

template<typename T>                // konzeptionelles Template
void func_for_cx(const T param);    // zum Ermitteln vom Typ von cx

func_for_cx(x);                     // konzeptioneller Aufruf: Ermittelter
                                    // Typ von param ist Typ von cx.

template<typename T>                // konzeptionelles Template
void func_for_rx(const T& param);   // zum Ermitteln vom Typ von rx

func_for_rx(x);                     // konzeptioneller Aufruf: Ermittelter
                                    // Typ von param ist Typ von rx.
```

Wie schon gesagt ist das Ableiten von Typen mit `auto` identisch mit dem für Templates – mit einer Ausnahme, auf die ich gleich eingehen werde.

Technik 1 unterteilt die Template-Typableitung in drei Fälle – basierend auf dem Charakter von *ParamType*, der Typ-Spezifikation für `param` im allgemeinen Funktions-Template. In einer Variablendeklaration mit `auto` nimmt die Typ-Spezifikation den Platz von *ParamType* ein, sodass es auch auch hier drei Fälle gibt:

- Fall 1: Die Typ-Spezifikation ist ein Zeiger oder eine Referenz, aber keine universelle Referenz.
- Fall 2: Die Typ-Spezifikation ist eine universelle Referenz.
- Fall 3: Die Typ-Spezifikation ist weder ein Zeiger noch eine Referenz.

Wir haben schon Beispiele für die Fälle 1 und 3 gesehen:

```
auto x = 27;          // Fall 3 (x ist weder Zeiger noch Referenz)

const auto cx = x;    // Fall 3 (cx auch nicht)

const auto& rx = x;   // Fall 1 (rx ist nicht-univers. Ref.)
```

Fall 2 arbeitet so, wie Sie es erwarten würden:

```
auto&& uref1 = x;     // x ist int und Lvalue,
                      // daher ist Typ von uref1 int&.

auto&& uref2 = cx;    // cx ist const int und Lvalue,
                      // daher ist Typ von uref2 const int&.

auto&& uref3 = 27;    // 27 ist int und Rvalue,
                      // daher ist Typ von uref3 int&&.
```

Technik 1 schließt mit einem Abschnitt, wie Array- und Funktionsnamen für nicht-referenzielle Typangaben zu Zeigern werden. Das passiert auch bei der auto-Typableitung:

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

auto arr1 = name;             // Typ von arr1 ist const char*.

auto& arr2 = name;            // Typ von arr2 ist
                              // const char (&)[13].


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

auto func1 = someFunc;        // Typ von func1 ist
                              // void (*)(int, double).

auto& func2 = someFunc;       // Typ von func2 ist
                              // void (&)(int, double).
```

Wie Sie sehen, geht die auto-Typableitung wie die Template-Typableitung vor. Es handelt sich letztendlich um zwei Seiten einer Medaille.

Mit einer Ausnahme. Beginnen wir mit der Beobachtung, dass C++98 beim Deklarieren eines int mit einem initialen Wert von 27 zwei Syntax-Varianten bietet:

```
int x1 = 27;
int x2(27);
```

C++11 ermöglicht aufgrund seiner Unterstützung für die vereinheitlichte Initialisierung noch diese:

```
int x3 = { 27 };
int x4{ 27 };
```

Insgesamt gibt es also vier Syntax-Varianten, aber nur ein Ergebnis: ein int mit dem Wert 27.

Wie aber in Technik 5 beschrieben wird, hat es Vorteile, Variablen über auto statt über explizite Typen zu deklarieren. Daher wäre es gut, int in den obigen Variablendeklarationen durch auto zu ersetzen. Ein einfaches Suchen und Ersetzen liefert diesen Code:

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

Diese Deklarationen lassen sich alle kompilieren, aber sie haben nicht die gleiche Bedeutung wie vor dem Ersetzen. Die ersten beiden Anweisungen deklarieren tatsächlich eine Variable vom Typ int mit dem Wert 27. Die zweiten beiden deklarieren hingegen eine Variable vom Typ std::initializer_list<int> mit einem einzelnen Element des Werts 27!

```
auto x1 = 27;            // Typ ist int, Wert ist 27.

auto x2(27);             // ebenso

auto x3 = { 27 };        // Typ ist std::initializer_list<int>,
                         // Wert ist { 27 }.

auto x4{ 27 };           // ebenso
```

Dies liegt an einer speziellen Typableitungsregel für auto. Ist der Initializer für eine per auto deklarierte Variable in geschweiften Klammern angegeben, ist der abgeleitete Typ eine std::initializer_list. Kann solch ein Typ nicht ermittelt werden (weil zum Beispiel die Werte im Braced Initializer unterschiedliche Typen haben), wird der Code nicht kompiliert:

```
auto x5 = { 1, 2, 3.0 };  // Fehler! T kann nicht für
                          // std::initializer_list<T>
                          // ermittelt werden.
```

Wie der Kommentar schon beschreibt, wird die Typableitung in diesem Fall nicht funktionieren, aber es ist wichtig, zu erkennen, dass hier tatsächlich zwei Arten von Typableitung beteiligt sind. Die eine resultiert aus dem Einsatz von auto: Der Typ von x5 muss abgeleitet werden. Da der Initializer von x5 in geschweiften Klammern angegeben ist, wird als Typ von x5 eine std::initializer_list ermittelt. std::initializer_list ist aber ein Template. Instanziierungen sind std::initializer_list<T> für einen Typ T, was bedeutet, dass der Typ von T ebenfalls abgeleitet werden muss. Das geschieht im Rahmen der Template-Typableitung. In diesem Beispiel geht das schief, weil die Werte im Braced Initializer keinen einheitlichen Typ besitzen.

Die Behandlung von Braced Initializers ist der einzige Punkt, in dem sich die Typableitung zwischen auto und Templates unterscheidet. Wird eine per auto deklarierte Variable mit einem Braced Initializer initialisiert, ist der abgeleitete Typ eine Instanziierung von std::initializer_list. Wird aber das entsprechende Template an den gleichen Initializer übergeben, schlägt die Ableitung fehl und der Code lässt sich nicht kompilieren:

```
auto x = { 11, 23, 9 };   // Typ von x ist
                          // std::initializer_list<int>.

template<typename T>      // Template mit Parameter-
void f(T param);          // deklaration entsprechend der
                          // Deklaration von x

f({ 11, 23, 9 });         // Fehler! Typ für T nicht ermittelbar.
```

Geben Sie allerdings im Template an, dass `param` für ein unbekanntes `T` vom Typ `std::initializer_list<T>` ist, wird die Template-Typableitung erkennen, was `T` ist:

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });         // T als int abgeleitet, Typ von initList
                          // ist std::initializer_list<int>.
```

Der einzige echte Unterschied zwischen der `auto`- und der Template-Typableitung ist also, dass `auto` *davon ausgeht*, dass ein Braced Initializer eine `std::initializer_list` repräsentiert, während das bei der Template-Typableitung nicht der Fall ist.

Sie fragen sich vielleicht, warum es für die Typableitung bei `auto` eine spezielle Regel für Braced Initializers gibt, nicht aber bei der Template-Typableitung. Das frage ich mich selbst. Ich habe bisher keine vernünftige Erklärung dafür gefunden. Aber die Regel ist nun einmal da, und Sie müssen daran denken, dass der abgeleitete Typ bei der Kombination aus `auto` und einem Braced Initializer immer `std::initializer_list` sein wird. Das ist besonders dann entscheidend, wenn Sie sich vorgenommen haben, die vereinheitlichte Initialisierung umzusetzen – wozu natürlich die Braced Initializers gehören. Ein klassischer Fehler in der C++11-Programmierung ist das unabsichtliche Deklarieren einer Variable vom Typ `std::initializer_list`, wenn Sie eigentlich einen anderen Typ haben wollen. Dieser Fallstrick ist einer der Gründe, warum manche Entwickler nur dann geschweifte Klammern um ihre Initialisierungsausdrücke legen, wenn sie es müssen. (Wann das der Fall ist, wird in Technik 7 erklärt.)

Bei C++11 ist das alles, aber bei C++14 geht noch mehr. C++14 ermöglicht es, mit `auto` den Rückgabetyp einer Funktion ableiten zu lassen (siehe Technik 3), und Lambdas in C++14 können `auto` bei der Parameterdeklaration verwenden. Allerdings wird bei diesen `auto`-Einsätzen die *Template-Typableitung* verwendet, nicht die `auto`-Typableitung. Eine Funktion mit einem `auto`-Rückgabetyp, die einen Braced Initializer zurückgibt, würde sich also nicht kompilieren lassen:

```
auto createInitList()
{
  return { 1, 2, 3 };         // Fehler: Typ nicht ableitbar
}                             // für { 1, 2, 3 }
```

Das Gleiche gilt, wenn `auto` in einer Parameter-Typ-Spezifikation bei einem C++14-Lambda eingesetzt wird:

```
std::vector<int> v;
…

auto resetV =
  [&v](const auto& newValue) { v = newValue; };     // C++14

…

resetV({ 1, 2, 3 });          // Fehler! Typ nicht ableitbar
                              // für { 1, 2, 3 }
```

Was Sie sich merken sollten

- Die auto-Typableitung entspricht meist der Template-Typableitung, allerdings geht die auto-Typableitung davon aus, dass ein Braced Initializer eine std::initializer_list repräsentiert, während das bei der Template-Typableitung nicht der Fall ist.
- auto in einem Rückgabetyp einer Funktion oder bei einem Lambda-Parameter nutzt die Template-Typableitung, nicht die auto-Typableitung.

Technik 3: Verstehen Sie decltype

decltype ist ein seltsames Ding. Geben Sie ihm einen Namen oder einen Ausdruck, gibt decltype Ihnen den Typ des Namens oder Ausdrucks aus. Dabei erhalten Sie im Allgemeinen genau das, was Sie erwarten würden. Gelegentlich aber erhalten Sie ein Ergebnis, bei dem Sie sich nur am Kopf kratzen können und erst einmal nachschlagen müssen, was das zu bedeuten hat.

Wir werden mit den normalen Fällen beginnen – denen, die keine Überraschungen liefern. Im Gegensatz zu dem, was während der Typableitung für Templates und auto passiert (siehe die Techniken 1 und 2), plappert decltype normalerweise genau den Typ des Namens oder Ausdrucks aus, den Sie ihm mitgeben:

```
const int i = 0;           // decltype(i) ist const int.

bool f(const Widget& w);   // decltype(w) ist const Widget&.
                           // decltype(f) ist bool(const Widget&).

struct Point {
  int x, y;                // decltype(Point::x) ist int.
};                         // decltype(Point::y) ist int.

Widget w;                  // decltype(w) ist Widget.

if (f(w)) …                // decltype(f(w)) ist bool.

template<typename T>       // vereinfachte Version von std::vector
```

```
class vector {
public:
  …
  T& operator[](std::size_t index);
  …
};

vector<int> v;              // decltype(v) ist vector<int>.
…
if (v[0] == 0) …            // decltype(v[0]) ist int&.
```

Sehen Sie? Keine Überraschungen.

In C++11 ist der wichtigste Einsatzbereich für `decltype` vermutlich das Deklarieren von Funktions-Templates, bei denen der Rückgabetyp der Funktion von den Parametertypen abhängt. Stellen Sie sich zum Beispiel vor, dass wir eine Funktion schreiben wollen, die einen Container übernimmt, der das Indexieren über eckige Klammern ermöglicht (also »`[]`«), dabei den Benutzer authentifiziert und das Ergebnis der Index-Operation zurückliefert. Der Rückgabetyp der Funktion sollte der gleiche sein wie der Typ der Index-Operation.

`operator[]` auf einem Container mit Objekten vom Typ `T` liefert im Allgemeinen ein `T&` zurück. Das gilt zum Beispiel für `std::deque` und ist so gut wie immer der Fall für `std::vector`. Für `std::vector<bool>` liefert `operator[]` allerdings kein `bool&`. Stattdessen erhalten Sie ein brandneues Objekt. Das Warum und Wie wird in Technik 6 beschrieben. Wichtig ist hier nur, dass der von `operator[]` für einen Container zurückgegebene Typ vom Container abhängt.

`decltype` macht das sehr einfach. Hier sehen Sie eine erste Version für das Template, das wir schreiben wollen. `decltype` berechnet dabei den Rückgabetyp. Das Template muss noch etwas verfeinert werden, aber das machen wir später:

```
template<typename Container, typename Index>    // funktioniert,
auto authAndAccess(Container& c, Index i)       // erfordert aber
  -> decltype(c[i])                             // Verbesserungen
{
  authenticateUser();
  return c[i];
}
```

Der Einsatz von `auto` vor dem Funktionsnamen hat nichts mit Typableitung zu tun, sondern mit der C++11-Syntax des *nachlaufenden Rückgabetyps*, bei dem der Rückgabetyp der Funktion nach der Parameterliste angegeben wird (nach dem »`->`«). Ein nachlaufender Rückgabetyp hat den Vorteil, dass die Parameter der Funktion in der Spezifikation des Rückgabetyps genutzt werden können. In `authAndAccess` spezifizieren wir zum Beispiel den Rückgabetyp anhand von `c` und `i`. Würde der Rückgabetyp ganz klassisch vor dem Funktionsnamen stehen, stünden `c` und `i` nicht zur Verfügung, da sie noch nicht deklariert wären.

Mit dieser Deklaration gibt `authAndAccess` den Typ zurück, den `operator[]` beim Anwenden auf den übergebenen Container liefert – also genau so, wie wir uns das vorgestellt haben.

C++11 ermöglicht es, den Rückgabetyp für Lambdas mit einer einzelnen Anweisung zu ermitteln, während C++14 dies für alle Lambdas und alle Funktionen erlaubt – auch für solche mit mehreren Anweisungen. Bei `authAndAccess` bedeutet das, dass wir in C++14 den nachlaufenden Rückgabetyp weglassen und nur `auto` nutzen können. Bei dieser Form des Einsatzes von `auto` kommt dann doch wieder die Typableitung ins Spiel. Insbesondere bedeutet dass, das der Compiler den Rückgabetyp der Funktion aus der Implementierung ableitet:

```
template<typename Container, typename Index>    // C++14;
auto authAndAccess(Container& c, Index i)       // nicht ganz
{                                               // korrekt
  authenticateUser();
  return c[i];                  // Rückgabetyp von c[i] abgeleitet
}
```

In Technik 2 ist beschrieben, dass Compiler für Funktionen, deren Rückgabetyp mit `auto` spezifiziert ist, die Template-Typableitung einsetzen. Das ist in diesem Fall problematisch. Wie ich erläutert habe, liefert `operator[]` für die meisten Container von `T` ein `T&` zurück, aber in Technik 1 ist erklärt, dass bei der Template-Typableitung die Referenzheit eines initialisierenden Ausdrucks ignoriert wird. Schauen wir uns an, was dies für den Client-Code bedeutet:

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10;  // Benutzer prüfen, d[5] zurück-
                           // geben, dann 10 zuweisen.
                           // Lässt sich nicht kompilieren!
```

Hier gibt `d[5]` ein `int&` zurück, aber die `auto`-Rückgabetypableitung für `authAndAccess` schneidet die Referenz ab und liefert als Rückgabetyp damit `int`. Dieses `int` ist als Rückgabetyp einer Funktion ein Rvalue, und der obige Code versucht dann, einem Rvalue-`int` den Wert 10 zuzuweisen. Das ist in C++ verboten, daher wird sich der Code nicht kompilieren lassen.

Damit `authAndAccess` so wie gewünscht funktioniert, müssen wir die `decltype`-Typableitung für den Rückgabetyp einsetzen, also festlegen, dass `authAndAccess` genau den Typ zurückgeben soll, den der Ausdruck `c[i]` liefert. Die Wächter von C++, denen durchaus bewusst war, dass die Regeln zur `decltype`-Typableitung manchmal zum Ableiten von Typen notwendig sind, ermöglichen dies in C++14 mithilfe der Angabe `decltype(auto)`. Das sieht auf den ersten Blick widersprüchlich aus (`decltype` *und* `auto`?), ist aber tatsächlich sinnvoll: `auto` legt fest, dass der Typ abgeleitet werden soll, während `decltype` angibt, dass zur Ableitung die `decltype`-Regeln zu nutzen sind. Wir können `authAndAccess` also so schreiben:

```
template<typename Container, typename Index>   // C++14; funktioniert,
decltype(auto)                                 // aber immer noch
authAndAccess(Container& c, Index i)           // Verbesserung
{                                              // notwendig
  authenticateUser();
  return c[i];
}
```

Jetzt gibt authAndAccess tatsächlich genau das zurück, was c[i] liefert. Insbesondere für den Normalfall, bei dem c[i] ein T& zurückgibt, liefert authAndAccess ebenfalls ein T&. Im eher ungewöhnlichen Fall, bei dem c[i] ein Objekt zurückgibt, liefert authAndAccess ebenfalls ein Objekt.

Der Einsatz von decltype(auto) ist nicht auf die Rückgabetypen von Funktionen beschränkt. Es kann auch beim Deklarieren von Variablen praktisch sein, wenn Sie die decltype-Regeln zur Typableitung auf den Initialisierungsausdruck anwenden wollen:

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;              // auto-Typableitung:
                                  // Typ von myWidget1 ist Widget.

decltype(auto) myWidget2 = cw;    // decltype-Typableitung:
                                  // Typ von myWidget2 ist
                                  //   const Widget&.
```

Aber zwei Dinge stören Sie noch, stimmt's? Eines ist das von mir erwähnte Verbessern von authAndAccess, auf das ich noch nicht eingegangen bin. Kümmern wir uns zuerst darum.

Schauen Sie sich nochmals die Deklaration für die C++14-Version von authAndAccess an:

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

Der Container wird als Lvalue-Referenz-auf-nicht-const übergeben, da die Rückgabe einer Referenz auf ein Element des Containers es dem Client ermöglicht, diesen Container zu verändern. Das bedeutet aber, dass es nicht möglich ist, Rvalue-Container an diese Funktion zu übergeben. Rvalues können nicht an Lvalue-Referenzen binden (sofern es nicht Lvalue-Referenzen-auf-const sind, was hier aber nicht der Fall ist).

Zugegeben, das Übergeben eines Rvalue-Containers an authAndAccess ist ein Sonderfall. Ein Rvalue-Container würde als temporäres Objekt typischerweise am Ende derjenigen Anweisung zerstört werden, die den Aufruf von authAndAccess enthält. Das bedeutet, dass eine Referenz auf ein Element in diesem Container (was authAndAccess im Allgemeinen zurückgibt) am Ende der Anweisung, die sie erzeugt hat, ins Leere zeigen würde. Es könnte trotzdem sinnvoll sein, ein temporäres Objekt an authAndAccess zu übergeben. Vielleicht will ein Client einfach eine Kopie eines Elements aus dem temporären Container erzeugen:

```
std::deque<std::string> makeStringDeque();   // Fabrikfunktion

// Kopie des 5. Elements der von
// makeStringDeque zurückgegebenen Deque machen
auto s = authAndAccess(makeStringDeque(), 5);
```

Wenn Sie solche Anwendungsfälle unterstützen, müssen Sie die Deklaration von authAndAccess so anpassen, dass sowohl Lvalues als auch Rvalues akzeptiert werden. Ein Überladen würde funktionieren (eine Version mit einem Lvalue-Referenzparameter, die andere

mit einem Rvalue-Referenzparameter), aber dann müssten wir zwei Funktionen warten. Sie können das vermeiden, indem `authAndAccess` einen Referenzparameter nutzt, der an Lvalues *und* Rvalues binden kann. In Technik 24 wird beschrieben, dass genau dafür universelle Referenzen da sind. `authAndAccess` kann daher so deklariert werden:

```
template<typename Container, typename Index>    // c ist jetzt
decltype(auto) authAndAccess(Container&& c,     // eine universelle
                             Index i);          // Referenz.
```

In diesem Template wissen wir nicht, mit was für einer Art Container wir arbeiten. Damit ignorieren wir auch den Typ des Index, mit dem wir darauf zugreifen können. Der Einsatz einer Pass-by-Value-Übergabe bei Objekten unbekannten Typs birgt das Risiko von Performance-Problemen durch unnötiges Kopieren, von Verhaltensproblemen beim Objekt-Slicing (siehe Technik 41) und dem Spott Ihrer Kollegen, aber im Fall von Container-Indizes scheint es vernünftig, sich die Standard-Library als Beispiel zu nehmen (zum Beispiel in `operator[]` für `std::string`, `std::vector` und `std::deque`) und eine Werteübergabe einzusetzen.

Wir müssen aber die Template-Implementierung anpassen, damit sie die Hinweise aus Technik 25 zum Anwenden von `std::forward` auf universelle Referenzen berücksichtigt:

```
template<typename Container, typename Index>       // finale
decltype(auto)                                    // C++14-
authAndAccess(Container&& c, Index i)             // Version
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

Damit sollte alles so laufen, wie wir es wollen, aber es wird ein C++14-Compiler benötigt. Haben Sie keinen, müssen Sie die C++11-Version des Templates einsetzen. Sie sieht wie ihr C++14-Gegenstück aus, Sie müssen aber den Rückgabetyp selbst angeben:

```
template<typename Container, typename Index>       // finale
auto                                              // C++11-
authAndAccess(Container&& c, Index i)             // Version
-> decltype(std::forward<Container>(c)[i])
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

Das andere Ding, das Sie sehr wahrscheinlich stört, ist mein Hinweis am Anfang dieser Technik, dass `decltype` *so gut wie immer* den erwarteten Typ liefert, aber *in seltenen Fällen* für Überraschungen sorgt. Sehr wahrscheinlich werden Sie solch einer seltenen Ausnahme nur über den Weg laufen, wenn Sie tagtäglich Bibliotheken implementieren.

Um das Verhalten von `decltype` *vollständig* zu verstehen, müssen Sie sich mit ein paar Spezialfällen vertraut machen. Die meisten davon sind zu abstrus, um sie in einem Buch wie diesem ernsthaft zu besprechen, aber ein Blick auf einen davon hilft sowohl beim Verstehen von `decltype` als auch bei seinem Einsatz.

Das Anwenden von `decltype` auf einen Namen liefert den deklarierten Typ für diesen Namen. Namen sind Lvalue-Ausdrücke, aber das hat keinen Einfluss auf das Verhalten von `decltype`. Für Lvalue-Ausdrücke, die komplizierter sind als Namen, stellt `decltype` allerdings sicher, dass der zurückgelieferte Typ immer eine Lvalue-Referenz ist. Hat also ein Lvalue-Ausdruck, der komplizierter als ein Name ist, den Typ `T`, liefert `decltype` den Typ als `T&`. Das hat selten eine Auswirkung, da der Typ der meisten Lvalue-Ausdrücke inhärent einen Lvalue-Referenz-Qualifier enthält. Funktionen geben zum Beispiel immer Lvalue-Referenzen zurück.

Es gibt aber eine Auswirkung dieses Verhaltens, derer man sich bewusst sein sollte. In

```
int x = 0;
```

ist `x` der Name einer Variable, daher liefert `decltype(x)` als Ergebnis `int`. Verpackt man den Namen `x` aber in Klammern »`(x)`«, erhält man einen Ausdruck, der komplexer als ein Name ist. Als Name ist `x` ein Lvalue, und C++ definiert den Ausdruck `(x)` ebenfalls als Lvalue. `decltype((x))` liefert daher `int&`. Allein durch das Umhüllen eines Namens mit Klammern ändert man also den Typ, den `decltype` zurückliefert!

In C++11 ist das nicht viel mehr als eine nette Kuriosität, aber zusammen mit der C++14-Unterstützung für `decltype(auto)` bedeutet dies, dass eine eigentlich triviale Änderung beim Schreiben einer `return`-Anweisung den abgeleiteten Typ einer Funktion beeinflussen kann:

```
decltype(auto) f1()
{
  int x = 0;
  …
  return x;        // decltype(x) ist int, f1 liefert int.
}

decltype(auto) f2()
{
  int x = 0;
  …
  return (x);      // decltype((x)) ist int&, f2 liefert int&.
}
```

Beachten Sie, dass `f2` nicht nur einen anderen Rückgabetyp als `f1` hat, es liefert auch noch eine Referenz auf eine lokale Variable! Mit solchem Code fahren Sie direkt ins Land des undefinierten Verhaltens – und da wollen Sie nicht hin.

Was lernen wir also daraus? Achten Sie genau darauf, was Sie beim Einsatz von `decltype(auto)` schreiben. Scheinbar unwichtige Änderungen am Ausdruck, dessen Typ abgeleitet werden soll, können diesen Typ verändern. Um sicherzustellen, dass der abgeleitete Typ der ist, den Sie erwarten, verwenden Sie die in Technik 4 beschriebenen Möglichkeiten.

Gleichzeitig sollten Sie sich bewusst sein, worum es hier geht. Sicher: `decltype` (sowohl allein als auch zusammen mit `auto`) liefert gelegentlich Überraschungen beim Ableiten des Typs, aber das passiert in normalen Situationen nicht. Üblicherweise gibt `decltype` genau

den gewünschten Typ zurück. Das gilt besonders dann, wenn `decltype` auf Namen angewendet wird, denn in dem Fall gereicht `decltype` seinem Namen zur Ehre: Es gibt den deklarierten Typ des Namens zurück.

Was Sie sich merken sollten

- `decltype` gibt so gut wie immer den Typ einer Variablen oder eines Ausdrucks ohne irgendwelche Modifikationen zurück.
- Für Lvalue-Ausdrücke des Typs `T`, die komplizierter sind als Namen, gibt `decltype` immer einen Typ `T&` zurück.
- C++14 bietet `decltype(auto)` an, das wie `auto` einen Typ aus seinem Initializer ableitet, dabei aber nach den `decltype`-Regeln vorgeht.

Technik 4: Zeigen Sie abgeleitete Typen an

Es hängt von der Phase des Softwareentwicklungsprozesses ab, woher Sie Ihre Informationen erhalten. Wir werden auf drei Möglichkeiten eingehen: Sie können Typinformationen beim Bearbeiten Ihres Codes, beim Kompilieren und zur Laufzeit erhalten.

IDE-Editoren

Codeeditoren in IDEs zeigen häufig die Typen von Programmentitäten an (Variablen, Parameter, Funktionen und so weiter), wenn Sie zum Beispiel den Mauszeiger über die Entität bewegen. Nehmen wir als Beispiel diesen Code:

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

Ein IDE-Editor würde sehr wahrscheinlich anzeigen, dass der für `x` abgeleitete Typ `int` und für `y` `const int*` ist.

Für solche Aktionen muss sich Ihr Code in einem mehr oder weniger kompilierbaren Zustand befinden, denn die IDE erhält diese Art von Informationen vom C++-Compiler (oder zumindest vom Frontend eines solchen), der innerhalb der IDE läuft. Wenn dieser Compiler Ihrem Code nicht genug Sinn entnehmen kann, um ihn zu parsen und die Typableitung durchzuführen, kann er die Typen auch nicht anzeigen.

Bei einfachen Typen wie `int` sind die Informationen der IDEs im Allgemeinen in Ordnung. Wie wir aber sehen werden, sind die Angaben der IDEs bei komplizierteren Typen schnell nicht mehr hilfreich.

Compiler-Diagnose

Am effektivsten können Sie einen Compiler einen Typ anzeigen lassen, indem Sie ihn so verwenden, dass der Compiler Probleme bekommt. Die Fehlermeldung wird dann mit großer Wahrscheinlichkeit den Typ beschreiben, der Ärger macht.

Stellen Sie sich zum Beispiel vor, dass wir aus den obigen Beispielen die Typen für `x` und `y` anzeigen lassen wollen. Zuerst deklarieren wir ein Klassen-Template, das wir *nicht definieren*. Das Folgende sollte ausreichen:

```
template<typename T>       // Nur Deklaration für TD;
class TD;                  // TD == "Type Displayer"
```

Versuche, dieses Template zu instanziieren, werden zu einer Fehlermeldung führen, da es keine Template-Definition gibt, die instanziiert werden könnte. Um die Typen für `x` und `y` ausgeben zu lassen, versuchen Sie einfach, `TD` mit deren Typen zu instanziieren:

```
TD<decltype(x)> xType;     // Fehler mit den Typen von
TD<decltype(y)> yType;     // x und y erzwingen
```

Ich verwende Variablennamen der Form *`variableName`*`Type`, da die so erzeugten Fehlermeldungen das Finden der Informationen erleichtern. Für den obigen Code hat einer meiner Compiler (unter anderem) folgende Meldungen ausgegeben (ich habe die Typinformationen hervorgehoben):

```
error: aggregate 'TD<int> xType' has incomplete type and
    cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type
    and cannot be defined
```

Ein anderer Compiler liefert die gleichen Informationen, nur in einer anderen Form:

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

Abgesehen von unterschiedlicher Formatierung haben alle Compiler, die ich ausprobiert habe, auf diese Weise Fehlermeldungen mit nützlichen Typinformationen geliefert.

Ausgabe zur Laufzeit

Die Ausgabe von Typinformationen mittels `printf` (nicht, dass ich Ihnen den Einsatz von `printf` empfehlen würde) ist nur zur Laufzeit möglich. Dafür haben Sie dann vollständige Kontrolle über das Ergebnis. Die Herausforderung besteht darin, eine sinnvolle textliche Repräsentation des Typs zu schaffen, für den Sie sich interessieren. »Kein Thema«, denken Sie sich, »da nehme ich einfach `typeid` und `std::type_info::name`.« Auf unserer andauernden Suche nach den für `x` und `y` abgeleiteten Typen meinen Sie vielleicht, folgenden Code schreiben zu können:

```
std::cout << typeid(x).name() << '\n';    // Typen für x und y
std::cout << typeid(y).name() << '\n';    // ausgeben
```

Bei diesem Vorgehen wird die Tatsache ausgenutzt, dass der Aufruf von `typeid` für ein Objekt wie `x` oder `y` ein Objekt vom Typ `std::type_info` zurückliefert, das eine Memberfunktion `name` besitzt. Diese erzeugt einen String im C-Stil (also einen `const char*`) mit dem Namen des Typs.

Aufrufe von `std::type_info::name` führen nicht zwingend zu etwas Sinnvollem, aber die Implementierungen versuchen hier, hilfreich zu sein. Allerdings variiert der Umfang der Hilfe. Die GNU- und Clang-Compiler geben zum Beispiel für den Typ von `x` den Wert »`i`« und für `y` den Wert »`PKi`« zurück. Diese Ergebnisse sind nützlich, sobald Sie den Code einmal entschlüsselt haben: »`i`« steht für »`int`«, »`PK`« für »Pointer to ~~`konst`~~ `const`.« (Beide Compiler unterstützen das Tool `c++filt`, das solche Kürzel aufdröselt.) Der Microsoft-Compiler erzeugt eine weniger kryptische Ausgabe: »`int`« für `x` und »`int const *`« für `y`.

Da diese Ergebnisse für die Typen von `x` und `y` korrekt sind, betrachten Sie das Typanzeige-Problem vielleicht als gelöst, aber seien Sie nicht zu voreilig. Schauen wir uns ein komplexeres Beispiel an:

```
template<typename T>                // Aufzurufende Template-
void f(const T& param);             // Funktion

std::vector<Widget> createVec();    // Fabrikfunktion

const auto vw = createVec();        // vw mit Ergebnis der Fabrikfunktion

if (!vw.empty()) {
  f(&vw[0]);                        // Aufruf von f
  …
}
```

Dieser Code enthält einen benutzerdefinierten Typ (`Widget`), einen STL-Container (`std::vector`) und eine `auto`-Variable (`vw`). Er dürfte deutlich näher an realen Situationen liegen, in denen Sie an den Typen interessiert sind, die der Compiler für Sie ableitet. So wäre es zum Beispiel nett zu wissen, welche Typen für den Template-Typparameter `T` und den Funktionsparameter `param` in `f` ermittelt wurden.

Lassen wir erst einmal `typeid` auf das Problem los. Ergänzen wir `f` einfach um etwas Code, um die fraglichen Typen auszugeben:

```
template<typename T>
void f(const T& param)
{
  using std::cout;

  cout << "T =     " << typeid(T).name() << '\n';     // T ausgeben

  cout << "param = " << typeid(param).name() << '\n'; // Ausgabe des
  …                                                  // Typs von
}                                                    // param
```

Die von den GNU- und Clang-Compilern erzeugten Executables liefern diese Ausgabe:

```
T =     PK6Widget
param = PK6Widget
```

Wir wissen schon, dass bei diesen Compilern PK für »Zeiger auf const« steht, daher ist das einzige Rätsel die Zahl 6. Dabei handelt es sich einfach um die Anzahl der Zeichen im folgenden Klassennamen (Widget). Diese Compiler sagen uns also, dass sowohl T als auch param vom Typ const Widget* sind.

Der Microsoft-Compiler schreibt dementsprechend:

```
T =     class Widget const *
param = class Widget const *
```

Drei verschiedene Compiler geben die gleichen Informationen aus, daher kann man davon ausgehen, dass das stimmt. Schauen Sie aber genau hin. Im Template f ist der für param deklarierte Typ const T&. Da scheint es doch seltsam, dass T und param vom gleichen Typ sind, oder? Würde T zum Beispiel vom Typ int sein, sollte der Typ von param doch const int& sein – also nicht genau der gleiche Typ.

Leider sind die Ergebnisse von std::type_info::name nicht zuverlässig. In diesem Fall ist zum Beispiel der von allen drei Compilern für param ausgegebene Typ falsch. Und die Compiler *müssen* sogar das falsche Ergebnis zurückliefern, weil laut Spezifikation von std::type_info::name der Typ so zu behandeln ist, als ob er als Werteparameter an eine Template-Funktion übergeben wurde. Wie ich in Technik 1 beschrieben habe, bedeutet das: Handelt es sich beim Typ um eine Referenz, wird diese ignoriert, und wenn der Typ nach dem Entfernen der Referenz const (oder volatile) ist, wird auch die constheit (oder volatileheit) ignoriert. Das ist der Grund, warum der Typ von param – const Widget * const & – als const Widget* ausgegeben wird. Erst wird die Referenzheit entfernt, dann die constheit des verbleibenden Zeigers.

Genauso schlecht ist, dass die von IDE-Editoren angezeigten Typinformationen ebenfalls nicht zuverlässig sind – oder zumindest nicht zuverlässig nützlich. Für das oben gezeigte Beispiel hat ein IDE-Editor angegeben, dass der Typ von T folgenden Wert hat (das habe ich mir nicht ausgedacht!):

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

Der gleiche IDE-Editor zeigt für den Typ von param:

```
const std::_Simple_types<...>::value_type *const &
```

Das ist weniger einschüchternd als der Typ von T, aber das »...« in der Mitte verwirrt nur so lange, bis Ihnen klar wird, dass der Editor damit sagen will: »Ich lasse all das Zeugs weg, das zum Typ von T gehört.« Mit ein bisschen Glück macht Ihre IDE ihre Aufgabe hier besser.

Wenn Sie sich lieber auf Bibliotheken als auf Ihr Glück verlassen, werden Sie sich freuen, dass die Boost TypeIndex-Bibliothek (oft geschrieben als *Boost.TypeIndex*) dann hilft, wenn std::type_info::name und die IDEs versagen. Die Bibliothek ist kein Teil des Standard-C++, aber IDEs oder Templates wie TD sind es auch nicht. Zudem stehen die Boost-Bibliotheken (zu finden unter *boost.org*) plattformübergreifend bereit, sind Open Source und haben eine Lizenz, die selbst den paranoidesten Firmenjuristen genehm sein

sollte – Code, der Boost-Bibliotheken einsetzt, kann also als nahezu so portabel angesehen werden wie Code, der die Standard-Library nutzt.

So kann unsere Funktion `f` exakte Typinformationen mithilfe von Boost.TypeIndex liefern:

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

template<typename T>
void f(const T& param)
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // T anzeigen
  cout << "T =     "
       << type_id_with_cvr<T>().pretty_name()
       << '\n';

  // Typ von param anzeigen
  cout << "param = "
       << type_id_with_cvr<decltype(param)>().pretty_name()
       << '\n';
  …
}
```

Das Funktions-Template `boost::typeindex::type_id_with_cvr` übernimmt dabei ein Typargument (den Typ, über den wir etwas erfahren wollen) und entfernt *nicht* `const`, `volatile` oder Referenz-Qualifier (daher das »`with_cvr`« im Template-Namen). Das Ergebnis ist ein Objekt vom Typ `boost::typeindex::type_index`, dessen Member-Funktion `pretty_name` einen `std::string` mit einer für Menschen lesbaren Darstellung des Typs erzeugt.

Mit dieser Implementierung von `f` schauen wir uns nochmals den Aufruf an, der beim Einsatz von `typeid` für `param` die falsche Typinformation geliefert hat:

```
std::vector<Widget> createVec();     // Fabrikfunktion

const auto vw = createVec();         // vm mit Ergebnis der Fabrikfunktion

if (!vw.empty()) {
  f(&vw[0]);                         // Aufruf von f
  …
}
```

Mit den Compilern von GNU und Clang liefert Boost.TypeIndex diese (korrekte) Ausgabe:

```
T =     Widget const*
param = Widget const* const&
```

Mit dem Microsoft-Compiler erhält man im Prinzip das Gleiche:

```
T =     class Widget const *
param = class Widget const * const &
```

Solche nahezu einheitlichen Ergebnisse sind zwar nett, aber denken Sie daran, dass IDE-Editoren, Compiler-Fehlermeldungen und Bibliotheken wie Boost.TypeIndex Ihnen nur dabei helfen können, herauszufinden, was Ihre Compiler für sich abgeleitet haben. Das mag nützlich sein, aber am besten ist es immer noch, die Typableitung zu verstehen. Diese ist in den Techniken 1–3 beschrieben.

Was Sie sich merken sollten

- Abgeleitete Typen können oft mit IDE-Editoren, über Compiler-Fehlermeldungen und die Boost TypeIndex-Library ausgegeben werden.
- Die Ergebnisse mancher dieser Tools sind eventuell weder hilfreich noch korrekt, daher müssen Sie sich mit den Typableitungsregeln von C++ auskennen.