HANSER

Leseprobe

Bjarne Stroustrup

Die C++-Programmiersprache

aktuell zum C++11-Standard

Übersetzt aus dem Englischen von Frank Langenau

ISBN (Buch): 978-3-446-43961-0

ISBN (E-Book): 978-3-446-43981-8

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43961-0

sowie im Buchhandel.

Inhalt

Vorwort

All problems in computer science
can be solved by another level of indirection,
except for the problem of too many layers of indirection.
– David J. Wheeler

C++ fühlt sich wie eine neue Sprache an. Das heißt, ich kann in C++11 meine Ideen klarer, einfacher und direkter ausdrücken als ich es in C++98 konnte. Darüber hinaus werden die Programme durch den Compiler besser überprüft und laufen auch schneller.

In diesem Buch strebe ich nach *Vollständigkeit*. Ich beschreibe alle Sprachfeatures und Komponenten der Standardbibliothek, die ein professioneller Programmierer wahrscheinlich braucht. Für jedes dieser Elemente gebe ich Folgendes an:

- *Grundprinzip:* Welche Probleme soll es lösen helfen? Welche Prinzipien liegen dem Design zugrunde? Was sind die grundsätzlichen Beschränkungen?
- *Spezifikation:* Wie ist es definiert? Dabei ist die Detailebene auf den erfahrenen Programmierer ausgerichtet; der angehende Sprachanwalt kann den vielen Verweisen zum ISO-Standard folgen.
- *Beispiele:* Wie kann man es an sich und in Kombination mit anderen Features sinnvoll einsetzen? Wie sehen die Schlüsseltechniken und Idioms aus? Welche Bedeutung hat es für Wartbarkeit und Performance?

Die Verwendung von C++ hat sich im Lauf der Jahre drastisch gewandelt und das Gleiche gilt auch für die Sprache selbst. Aus dem Blickwinkel eines Programmierers sind die meisten Änderungen als Verbesserungen einzustufen. Das aktuelle ISO-Standard-C++ (ISO/IEC 14882-2011, üblicherweise C++11 genannt) ist ein weit besseres Tool für Qualitätssoftware als die vorherigen Versionen. Worin äußert sich dies? Welche Programmierstile und -techniken unterstützt modernes C++? Welche Features der Sprache und der Standardbibliothek unterstützen diese Techniken? Was sind die Bausteine von elegantem, korrektem, wartbarem und effizientem C++-Code? Dies sind die Schlüsselfragen, die in diesem Buch beantwortet werden. Viele Antworten sind nicht die gleichen, wie Sie sie für Vintage-C++ der Jahre 1985, 1995 oder 2005 finden: Der Fortschritt ist nicht aufzuhalten.

C++ ist eine universelle Programmiersprache, die den Entwurf und die Verwendung von typreichen, kompakten Abstraktionen betont. Die Sprache ist besonders für ressourcenbeschränkte Anwendungen geeignet, wie man sie beispielsweise in Softwareinfrastrukturen findet. Es lohnt sich für den Programmierer, Zeit in das Erlernen der Techniken für qualitätsgerechten Code zu investieren. C++ ist eine Sprache für jemanden, der Programmierung ernst nimmt. Unsere Zivilisation hängt entscheidend von Software ab; es wäre also besser, wenn es sich dabei um Qualitätssoftware handelt.

Es gibt Milliarden von Zeilen in C++-Code. Dabei wird besonderer Wert auf Stabilität gelegt, sodass der C++-Code von 1985 und 1995 immer noch funktioniert und auch für weitere Jahrzehnte funktionieren wird. Allerdings können Sie mit modernem C++ besser arbeiten; wenn Sie sich an die älteren Stile klammern, werden Sie Code von geringerer Qualität und schlechterer Leistung schreiben. Die Betonung von Stabilität bedeutet auch, dass standardkonformer Code, den Sie heute schreiben, auch noch in einigen Jahrzehnten funktionieren wird. Der gesamte Code in diesem Buch ist zum 2011-ISO-C++-Standard konform.

Dieses Buch richtet sich an drei Leserkreise:

- C++-Programmierer, die wissen möchten, was der neueste ISO-C++-Standard zu bieten hat,
- C-Programmierer, die daran interessiert sind, was C++ über C hinaus bietet, und
- Programmierer, die von Anwendungssprachen wie zum Beispiel Java, C#, Python und Ruby kommen und nach etwas suchen, das „näher an der Maschine“ dran ist – etwas, was flexibler ist, was eine bessere Prüfung zur Übersetzungszeit realisiert oder was eine bessere Performance bietet.

Natürlich gibt es keine klare Trennung zwischen diesen drei Gruppen – ein professioneller Softwareentwickler beherrscht schließlich mehr als nur eine Programmiersprache.

In diesem Buch wird davon ausgegangen, dass die Leser Programmierer sind. Wenn Sie fragen: „Was ist eine **for**-Schleife?“ oder „Was ist ein Compiler?“, dann ist dieses Buch (noch) nichts für Sie; stattdessen empfehle ich mein *„Principles and Practice Using C++ to get started with programming and C++“*. Darüber hinaus nehme ich an, dass die Leser bereits eine gewisse Reife als Softwareentwickler haben. Wenn Sie fragen: „Warum sich mit Testen abmühen?“ oder sagen „Alle Sprachen sind prinzipiell gleich; zeige mir nur die Syntax“, vielleicht auch überzeugt davon sind, dass es genau eine Sprache gibt, die für sämtliche Aufgaben ideal ist, dann ist dieses Buch ebenfalls nichts für Sie.

Welche Features bietet C++11 gegenüber C++98 und darüber hinaus? Ein Maschinenmodell, das für moderne Computer mit jeder Menge Parallelität geeignet ist. Sprach- und Standardbibliotheksinstrumente für nebenläufige Programmierung auf Systemniveau (z.B. mithilfe von Mehrkernprozessoren). Verarbeitung regulärer Ausdrücke, Ressourcenverwaltungszeiger, Zufallszahlen, verbesserte Container (einschließlich Hashtabellen) und vieles mehr. Allgemeine und einheitliche Initialisierung, eine einfachere **for**-Anweisung, Verschiebesemantik, grundlegende Unicode-Unterstützung, Lambda-Ausdrücke, allgemeine konstante Ausdrücke, Kontrolle über Standardwerte von Klassen, variadische Templates, benutzerdefinierte Literale und mehr. Denken Sie bitte daran, dass diese Bibliotheken und Sprachfeatures dafür da sind, Programmiertechniken für die Entwicklung von Qualitätssoftware zu unterstützen. Kombinieren Sie diese – wie Bausteine aus einem Baukasten –, um ein konkretes Problem zu lösen, anstatt sie einzeln in relativer Isolation zu verwenden. Ein Computer ist eine universelle Maschine und mit C++ machen Sie dessen Kapazität nutzbar. Insbesondere ist C++ konzeptionell darauf ausgelegt, genügend flexibel und allgemein zu sein, um mit zukünftigen Problemen umgehen zu können, von denen die Designer der Sprache noch nicht einmal geträumt haben.

Danksagung

Außer den Leuten, die in den Danksagungen der vorherigen Ausgaben erwähnt wurden, möchte ich Pete Becker, Hans-J. Boehm, Marshall Clow, Jonathan Coe, Lawrence Crowl, Walter Daugherty, J. Daniel Garcia, Robert Harle, Greg Hickman, Howard Hinnant, Brian Kernighan, Daniel Krügler, Nevin Liber, Michel Michaud, Gary Powell, Jan Christiaan van Winkel und Leor Zolman danken. Ohne ihre Hilfe wäre dieses Buch mit Sicherheit nicht so gut geworden.

Dank auch an Howard Hinnant für die Beantwortung vieler Fragen zur Standardbibliothek.

Andrew Sutton ist der Autor der Origin-Bibliothek, die als Testumgebung für die Emulation von Konzepten in den Template-Kapiteln dient, und der Matrix-Bibliothek, die Thema von Kapitel 29 ist. Die Open-Source-Bibliothek Origin finden Sie im Web, wenn Sie nach „Origin“ und „Andrew Sutton“ suchen.

Ein Dank geht auch an meine Design-Diplomanden, die mehr Probleme mit den „Tour-Kapiteln“ als sonst jemand gefunden haben.

Hätte ich sämtlichen Hinweisen meiner Rezensenten folgen können, wäre das Buch noch besser, doch auch Hunderte Seiten länger geworden. Jeder Fachlektor hat zusätzliche technische Details vorgeschlagen, erweiterte Beispiele und viele nützliche Entwicklungskonventionen; von jedem neuen Rezensenten (oder Lehrer) kamen Vorschläge für ergänzende Beispiele; und viele Rezensenten haben (zu Recht) eingeschätzt, dass das Buch zu umfangreich werden könnte.

Danken möchte ich der Computer Science Department der Princeton University und speziell Prof. Brian Kernighan, die mich für einen Teil des Sabbaticals aufgenommen haben, was mir Zeit verschaffte, dieses Buch zu schreiben. Aus dem gleichen Grund sei dem Computer Lab der Cambridge University und speziell Prof. Andy Hopper gedankt.

College Station, Texas *Bjarne Stroustrup*

26 Instanziierung

For every complex problem,
there is an answer that is
clear, simple, and wrong.
– H. L. Mencken

26.1 Einführung

Zu den größten Stärken von Templates zählt ihre Flexibilität, mit der sich Code arrangieren lässt. Der Compiler kombiniert Code (Informationen) aus

- der Template-Definition und ihrer lexikalischen Umgebung,
- den Template-Argumenten und ihrer lexikalischen Umgebung und
- der Einsatzumgebung des Templates,

um eine eindrucksvolle Codequalität zu produzieren. Der Schlüssel zur resultierenden Performance liegt darin, dass sich der Compiler den Code aus diesen Umgebungen gleichzeitig ansehen und ihn anhand aller verfügbaren Informationen verflechten kann. Das Problem dabei ist, dass Code in einer Template-Definition nicht so örtlich begrenzt ist, wie wir es (unter sonst gleichen Umständen) gern hätten. Manchmal ist nicht ganz klar, worauf sich ein Name, der in einer Template-Definition verwendet wird, bezieht:

- Ist es ein lokaler Name?
- Ist es ein Name, der mit einem Template-Argument verbunden ist?
- Ist es ein Name von einer Basisklasse in einer Hierarchie?
- Ist es ein Name aus einem benannten Namespace?
- Ist es ein globaler Name?

Dieses Kapitel diskutiert Fragen, die sich auf *Namensbindung* beziehen, und betrachtet die Konsequenzen für Programmierstile.

- Templates wurden in §3.4.1 und §3.4.2 eingeführt.
- Kapitel 23 gibt eine ausführliche Einführung in Templates und die Verwendung von Template-Argumenten.
- Kapitel 24 führt generische Programmierung und den Kerngedanken der Konzepte ein.
- Kapitel 25 beschäftigt sich mit Details von Klassen-Templates und Funktions-Templates und führt den Begriff der Spezialisierung ein.
- Kapitel 27 diskutiert die Beziehung zwischen Templates und Klassenhierarchien (die generische und objektorientierte Programmierung unterstützen).

- Kapitel 28 stellt Templates als Sprache für das Generieren von Klassen und Funktionen in den Mittelpunkt.
- Kapitel 29 präsentiert ein größeres Beispiel, wie sich die Sprachmittel und Programmiertechniken kombiniert einsetzen lassen.

26.2 Template-Instanziierung

Für eine gegebene Template-Definition und eine Verwendung dieses Templates ist es Aufgabe der Implementierung, korrekten Code zu generieren. Aus einem Klassen-Template und einem Satz von Template-Argumenten muss der Compiler die Definition einer Klasse und die Definitionen ihrer Member-Funktionen, die im Programm verwendet werden (und nur diese; §26.2.1) generieren. Aus einer Template-Funktion und einem Satz von Template-Argumenten muss eine Funktion generiert werden. Dieser Vorgang ist die sogenannte *Template-Instanziierung*.

Die generierten Klassen und Funktionen heißen *Spezialisierungen*. Wenn wir zwischen generierten Spezialisierungen und explizit durch den Programmierer geschriebenen Spezialisierungen (§25.3) unterscheiden müssen, sprechen wir von *generierten Spezialisierungen* bzw. *expliziten Spezialisierungen*. Eine explizite Spezialisierung wird oftmals auch als *benutzerdefinierte Spezialisierung* oder einfach *Benutzerspezialisierung* bezeichnet.

Um Templates in nichttrivialen Programmen zu verwenden, muss ein Programmierer die Grundlagen beherrschen, wie Namen, die in einer Template-Definition erscheinen, an Deklarationen gebunden werden und wie sich der Quellcode organisieren lässt (§23.7).

Standardmäßig generiert der Compiler Klassen und Funktionen aus den verwendeten Templates entsprechend den Regeln für Namensbindung (§26.3). Das heißt, ein Programmierer muss nicht explizit angeben, welche Versionen von welchen Templates generiert werden müssen. Dies ist auch sinnvoll, weil der Programmierer gar nicht ohne Weiteres genau wissen kann, welche Versionen eines Templates erforderlich sind. Oftmals werden Templates, von denen der Programmierer nie etwas gehört hat, in der Implementierung von Bibliotheken verwendet, und manchmal werden Templates, die dem Programmierer bekannt sind, mit unbekannten Template-Argumenttypen verwendet. Zum Beispiel ist `map` (§4.4.3, §31.4.3) der Standardbibliothek in Form eines Rot-Schwarz-Baums als Template mit Datentypen und Operationen implementiert, die außer vielleicht einem neugierigen Benutzer praktisch niemandem bekannt sind. Im Allgemeinen lässt sich der erforderliche Satz der generierten Funktionen nur in Erfahrung bringen, wenn man die in Codebibliotheken verwendeten Templates rekursiv untersucht. Für derartige Analysen sind Computer besser geeignet als der Mensch.

Andererseits ist es für einen Programmierer manchmal wichtig, spezifisch festlegen zu können, wo Code aus einem Template generiert werden soll (§26.2.2). Auf diese Weise kann der Programmierer den Kontext der Instanziierung detailliert steuern.

26.2.1 Wann wird Instanziierung gebraucht?

Eine Spezialisierung eines Klassen-Templates muss nur generiert werden, wenn die Definition der Klasse gebraucht wird (§iso.14.7.1). Um speziell einen Zeiger auf eine bestimmte Klasse zu deklarieren, ist die eigentliche Definition einer Klasse nicht erforderlich. Zum Beispiel:

```
class X;
X* p;        // OK: keine Definition von X erforderlich
X a;         // Fehler: Definition von X notwendig
```

Diese Unterscheidung kann ausschlaggebend sein, wenn man Template-Klassen definiert. Eine Template-Klasse wird *nicht* instanziiert, außer wenn ihre Definition tatsächlich benötigt wird. Zum Beispiel:

```
template<typename T>
class Link {
    Link* suc;        // OK: Definition von Link (noch) nicht erforderlich
    // ...
};

Link<int>* pl;        // Instanziierung von Link<int> (noch) nicht erforderlich

Link<int> lnk;        // jetzt müssen wir Link<int> instanziieren
```

Ein Platz, wo ein Template verwendet wird, definiert einen Punkt der Instanziierung (§26.3.3).

Eine Implementierung instanziiert eine Template-Funktion nur, wenn diese Funktion verwendet wurde. Mit „verwendet“ meinen wir „aufgerufen oder ihre Adresse ermitteln lassen“. Insbesondere zieht die Instanziierung eines Klassen-Templates nicht die Instanziierung aller seiner Member-Funktionen nach sich. Damit ist der Programmierer sehr flexibel, wenn er eine Template-Klasse definiert. Sehen Sie sich dazu folgenden Code an:

```
template<typename T>
class List {
    // ...
    void sort();
};

class Glob {
    // ... keine Vergleichsoperatoren ...
};

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // ... Operationen auf lb, aber nicht lb.sort() verwenden ...
}
```

Hier wird **`List<string>::sort()`** instanziiert, **`List<Glob>::sort()`** jedoch nicht. Dies verringert die Menge des generierten Codes und erspart es uns, das Programm neu entwerfen zu müssen. Wäre **`List<Glob>::sort()`** generiert worden, müssten wir entweder die von **`List::sort()`** benötigten Operationen zu **`Glob`** hinzufügen, die Funktion **`sort()`** redefinieren, damit sie kein Member mehr von **`List`** ist (ohnehin das bessere Design), oder einen anderen Container für **`Glob`**-Typen verwenden.

26.2.2 Manuelle Kontrolle der Instanziierung

Die Sprache setzt für eine Template-Instanziierung keine explizite Benutzeraktion voraus. Allerdings bietet sie zwei Mechanismen, damit der Benutzer bei Bedarf die Kontrolle übernehmen kann. Denn manchmal möchte er

- den Vorgang beim Übersetzen und Binden optimieren, indem redundante Instanziierungen eliminiert werden, oder
- genau wissen, welcher Punkt der Instanziierung verwendet wird, um Überraschungen zu vermeiden, die sich aus komplizierten Umgebungen bei der Namensbindung ergeben.

Eine explizite Instanziierungsanforderung (oftmals einfach *explizite Instanziierung* genannt) ist eine Deklaration einer Spezialisierung, die mit dem Schlüsselwort **`template`** als Präfix (ohne nachfolgendes **`<`**) versehen ist:

```
template class vector<int>;                    // Klasse
template int& vector<int>::operator[](int);    // Member-Funktion
template int convert<int,double>(double);      // Nicht-Member-Funktion
```

Eine Template-Deklaration beginnt mit **`template<`**, während ein reines **`template`** eine Instanziierungsanforderung einleitet. Beachten Sie, dass **`template`** als Präfix vor einer vollständigen Deklaration steht; es genügt nicht, nur einen Namen anzugeben:

```
template vector<int>::operator[];    // Syntaxfehler
template convert<int,double>;        // Syntaxfehler
```

Wie in Aufrufen von Template-Funktionen können die Template-Argumente, die aus den Funktionsargumenten hergeleitet werden können, entfallen (§23.5.1). Zum Beispiel:

```
template int convert<int,double>(double);    // OK (redundant)
template int convert<int>(double);           // OK
```

Wird ein Klassen-Template explizit instanziiert, wird auch jede Member-Funktion instanziiert.

Die Instanziierungsanforderungen können sich erheblich auf die Bindungszeit und die Effizienz der Neukompilierung auswirken. Ich habe Beispiele gesehen, in denen ein Bundling der meisten Template-Instanziierungen zu einer einzigen Übersetzungseinheit die Übersetzungszeit von mehreren Stunden auf einige Minuten gedrückt hat.

Wenn zwei Definitionen für dieselbe Spezialisierung existieren, ist das ein Fehler. Es spielt keine Rolle, ob solche Mehrfachspezialisierungen benutzerdefiniert sind (§25.3), implizit generiert (§23.2.2) oder explizit angefordert werden. Allerdings ist ein Compiler nicht verpflichtet, mehrere Instanziierungen in getrennten Übersetzungseinheiten zu diagnostizieren. Eine intelligente Implementierung kann damit redundante Instanziierungen ignorieren und so Probleme vermeiden, die mit der Komposition von Programmen aus Bibliotheken mithilfe expliziter Instanziierung zusammenhängen. Implementierungen müssen aber nicht unbedingt intelligent sein. Benutzer von „weniger intelligenten“ Implementierungen müssen Mehrfachinstanziierungen vermeiden. Wenn sie dies nicht tun, lässt sich ihr Programm im ungünstigsten Fall nicht binden; stillschweigende Bedeutungsänderungen gibt es nicht.

Als Ergänzung zu expliziten Instanziierungsanforderungen bietet die Sprache explizite Anforderungen, *nicht* zu instanziieren (normalerweise **`extern template`**s genannt). Diese Option bietet sich an, wenn eine Spezialisierung explizit zu instanziieren und ihre Instanziierung in anderen Übersetzungseinheiten mit **`extern template`** zu unterdrücken ist. Dies spiegelt das klassische Paradigma einer Definition und vieler Deklarationen (§15.2.3) wider. Zum Beispiel:

```
#include "MyVector.h"

extern template class MyVector<int>;    // unterdrückt implizite Instanziierung
                                        // an anderer Stelle explizit instanziieren
void foo(MyVector<int>& v)
{
    // ... den Vektor hier verwenden ...
}
```

Das „an anderer Stelle“ könnte etwa so aussehen:

```
#include "MyVector.h"

template class MyVector<int>;    // in dieser Übersetzungseinheit instanziieren;
                                 // diesen Punkt der Instanziierung verwenden
```

Außer Spezialisierungen für alle Member einer Klasse zu generieren, bestimmt die explizite Instanziierung auch einen einzelnen Punkt der Instanziierung, sodass andere Punkte der Instanziierung (§26.3.3) ignoriert werden können. Diese nutzt man beispielsweise, um explizite Instanziierung in einer gemeinsamen Bibliothek zu platzieren.

26.3 Namensbindung

Definieren Sie Template-Funktionen, um Abhängigkeiten von nichtlokalen Informationen zu minimieren. Denn Templates sind dafür vorgesehen, Funktionen und Klassen basierend auf unbekannten Typen und unbekannten Kontexten zu generieren. Jede unauffällige Kontextabhängigkeit zeigt sich wahrscheinlich als Problem – und zwar für den Programmierer, der eigentlich gar nicht an den Implementierungsdetails des Templates interessiert ist. Die allgemeine Regel, globale Namen weitestgehend zu vermeiden, sollte in Template-Code besonders ernst genommen werden. Dementsprechend versuchen wir, Template-Definitionen möglichst eigenständig abzufassen und so viel wie möglich von dem, was andernfalls im globalen Kontext erscheinen würde, in Form von Template-Argumenten bereitzustellen (z. B. Traits; §28.2.4, §33.1.3). Verwenden Sie Konzepte, um Abhängigkeiten von Template-Argumenten zu dokumentieren (§24.3).

Allerdings ist es durchaus üblich, dass bestimmte nichtlokale Namen verwendet werden müssen, um die eleganteste Formulierung eines Templates zu erreichen. So schreibt man häufiger einen Satz von kooperierenden Template-Funktionen als lediglich nur eine selbstständige Funktion. Derartige Funktionen können manchmal Klassen-Member sein, sind es jedoch nicht immer. Manchmal sind nichtlokale Funktionen die beste Wahl. Typische Beispiele dafür sind die **`swap()`**- und **`less()`**-Aufrufe von **`sort()`** (§25.3.4). Die Algorithmen der Standardbibliothek verkörpern ein Beispiel im großen Stil (Kapitel 32). Wenn etwas nicht-

lokal sein muss, bevorzugen Sie einen benannten Namespace gegenüber dem globalen Gültigkeitsbereich. Damit wird auch eine gewisse Lokalität bewahrt.

Operationen mit konventionellen Namen und konventioneller Semantik wie zum Beispiel **+**, *****, **[]** und **sort()** sind eine weitere Quelle für nichtlokale Namen, die in einer Template-Definition verwendet werden. Sehen Sie sich dazu folgenden Code an:

```
bool tracing;

template<typename T>
T sum(std::vector<T>& v)
{
    T t {};
    if (tracing)
        cerr << "sum(" << &v << ")\n";
    for (int i = 0; i!=v.size(); ++i)
        t = t + v[i];
    return t;
}
// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
    Quad c = sum(v);
}
```

Die harmlos aussehende Template-Funktion **sum()** hängt von mehreren Namen ab, die in ihrer Definition nicht explizit angegeben sind, wie zum Beispiel **tracing**, **cerr** und dem Operator **+** . In diesem Beispiel ist **+** in **<quad.h>** definiert:

```
Quad operator+(Quad,Quad);
```

Vor allem aber befindet sich nichts, was sich auf **Quad** bezieht, im Gültigkeitsbereich, wenn **sum()** definiert wird, und beim Verfasser von **sum()** kann man nicht davon ausgehen, dass er die Klasse **Quad** kennt. So kann der Operator **+** im Programmtext erst hinter **sum()** und sogar zu einem späteren Zeitpunkt definiert werden. Die sogenannte *Namensbindung* sucht die Deklaration für jeden Namen, der explizit oder implizit in einem Template verwendet wird. Allgemein haftet der Template-Namensbindung das Problem an, dass drei Kontexte an einer Template-Instanziierung beteiligt sind und sich nicht sauber trennen lassen:

1. Der Kontext der Template-Definition
2. Der Kontext der Argumenttypdeklaration
3. Der Kontext, in dem das Template verwendet wird

Wenn wir ein Funktions-Template definieren, sollte genügend Kontext verfügbar sein, damit die Template-Definition in Form ihrer tatsächlichen Argumente sinnvoll sein kann, ohne am Punkt der Verwendung „versehentlich“ etwas aus der Umgebung aufzugreifen. Um uns dabei zu helfen, trennt die Sprache Namen, die in einer Template-Definition verwendet werden, in zwei Kategorien:

1. *Abhängige Namen:* Namen, die von einem Template-Parameter abhängen. Derartige Namen werden am Punkt der Instanziierung (§26.3.3) gebunden. Im **sum()**-Beispiel

kann die Definition des Operators `+` im Instanziierungskontext gefunden werden, weil er Operanden des Template-Argumenttyps übernimmt.

2. *Unabhängige Namen:* Namen, die nicht von einem Template-Parameter abhängen. Derartige Namen werden am Punkt der Definition des Templates (§26.3.2) gebunden. Im `sum()`-Beispiel ist das Template `vector` im Standard-Header `<vector>` definiert und der boolesche Wert `tracing` befindet sich im Gültigkeitsbereich, wenn der Compiler auf die Definition von `sum()` trifft.

Sowohl abhängige als auch unabhängige Namen müssen sich entweder am Punkt der Verwendung im Gültigkeitsbereich befinden oder über eine argumentabhängige Suche (Argument-Dependent Lookup, ADL; §14.2.4) auffindbar sein, um berücksichtigt zu werden. Für umfassende Details sei auf §iso.14.6 verwiesen.

26.3.1 Abhängige Namen

Die einfachste Definition von „`N` hängt von einem Template-Parameter `T` ab“ wäre „`N` ist ein Member von `T`“. Leider genügt das nicht ganz; die Addition von `Quad`-Werten (§26.3) ist ein Gegenbeispiel. Folglich betrachtet man einen Funktionsaufruf nur dann als von einem Template-Argument *abhängig*, wenn eine dieser Bedingungen gilt:

1. Der Typ des tatsächlichen Arguments hängt von einem Template-Parameter `T` entsprechend den Regeln der Typherleitung (§23.5.2) ab, zum Beispiel `f(T(1))`, `f(t)`, `f(g(t))` und `f(&t)`, unter der Annahme, dass `t` ein `T` ist.
2. Die aufgerufene Funktion hat einen Parameter, der von `T` entsprechend den Regeln der Typherleitung (§23.5.2) abhängt, zum Beispiel `f(T)`, `f(list<T>&)` und `f(const T*)`.

Der Name einer aufgerufenen Funktion ist prinzipiell dann abhängig, wenn die Abhängigkeit ohne Weiteres anhand ihrer Argumente oder formaler Parameter erkennbar ist. Zum Beispiel:

```
template<typename T>
int f(T a)
{
    return g(a);    // OK: a ist ein abhängiger Name und demzufolge auch g
}

class Quad { /* ... */ };
int g(Quad);

int z = f(Quad{2});        // g von f ist an g(Quad) gebunden
```

Ein Aufruf, bei dem ein Argument zufällig dem Typ eines tatsächlichen Template-Parameters entspricht, ist nicht abhängig. Zum Beispiel:

```
class Quad { /* ... */ };

template<typename T>
int ff(T a)
{
    return gg(Quad{1});        // Fehler: kein gg() im Gültigkeitsbereich und
                               // gg(Quad{1}) hängt nicht von T ab
}
```

```
int gg(Quad);

int zz = ff(Quad{2});
```

Würde die Funktion **gg(Quad{1})** als abhängig betrachtet, wäre ihre Bedeutung für einen Leser der Template-Definition höchst mysteriös. Wenn ein Programmierer möchte, dass **gg(Quad)** aufgerufen wird, sollte die Deklaration von **gg(Quad)** vor der Definition von **ff()** erscheinen, sodass sich **gg(Quad)** im Gültigkeitsbereich befindet, wenn **ff()** analysiert wird. Dies ist genau die gleiche Regel, die auch für Definitionen von Nicht-Template-Funktionen (§26.3.2) gilt.

Standardmäßig wird von einem abhängigen Namen angenommen, dass er etwas benennt, was kein Typ ist. Um also einen abhängigen Namen als Typ zu verwenden, müssen Sie dies mit dem Schlüsselwort **typename** ausdrücken. Zum Beispiel:

```
template<typename Container>
void fct(Container& c)
{
    Container::value_type v1 = c[7];    // Syntaxfehler: value_type wird als Name
                                        // eines Nicht-Typs angenommen
    typename Container::value_type v2 = c[9];    // OK: value_type benennt
                                        // vermutlich einen Typ
    auto v3 = c[11];                    // OK: Typ vom Compiler herausfinden lassen
    // ...
}
```

Wir können dieses umständliche **typename** vermeiden, wenn wir einen Typalias einführen (§23.6). Zum Beispiel:

```
template<typename T>
using Value_type = typename T::value_type;

template<typename Container>
void fct2(Container& c)
{
    Value_type<Container> v1 = c[7];    // OK
    // ...
}
```

Analog dazu müssen wir bei der Benennung eines Member-Templates nach einem **.** (Punkt), **->** oder **::** das Schlüsselwort **template** angeben. Zum Beispiel:

```
class Pool {    // ein Allokator
public:
    template<typename T> T* get();
    template<typename T> void release(T*);
    // ...
};

template<typename Alloc>
void f(Alloc& all)
{
    int* p1 = all.get<int>();             // Syntaxfehler: für get wird angenommen,
                                          // dass ein Nicht-Template benannt wird
    int* p2 = all.template get<int>();    // OK: für get() wird angenommen,
```

```
                                        // dass ein Template benannt wird
    // ...
}

void user(Pool& pool)
{
    f(pool);
    // ...
}
```

Verglichen mit **typename** (um explizit auszudrücken, dass ein Name einen Typ benennt) ist es eher selten, mit **template** explizit auszudrücken, dass ein Name ein Template benennt. Beachten Sie die unterschiedlichen Positionen des Schlüsselworts, das die Mehrdeutigkeit beseitigt: **typename** erscheint vor dem qualifizierten Namen und **template** unmittelbar vor dem Template-Namen.

26.3.2 Bindung am Punkt der Definition

Wenn der Compiler eine Template-Definition sieht, ermittelt er, welche Namen abhängig sind (§26.3.1). Ist ein Name abhängig, wird die Suche nach seiner Deklaration auf den Zeitpunkt der Instanziierung verschoben (§26.3.3).

Namen, die nicht von einem Template-Argument abhängen, werden wie Namen behandelt, die nicht in Templates vorkommen; sie müssen sich am Punkt der Definition im Gültigkeitsbereich (§6.3.4) befinden. Zum Beispiel:

```
int x;

template<typename T>
T f(T a)
{
    ++x;         // OK: x ist im Gültigkeitsbereich
    ++y;         // Fehler: kein y im Gültigkeitsbereich und y hängt nicht von T ab
    return a;    // OK: a ist abhängig
}

int y;

int z = f(2);
```

Eine einmal gefundene Deklaration wird auch verwendet, selbst wenn später vielleicht eine „bessere“ Deklaration gefunden werden könnte. Zum Beispiel:

```
void g(double);
void g2(double);

template<typename T>
int ff(T a)
{
    g2(2);    // g2(double) aufrufen
    g3(2);    // Fehler: kein g3() im Gültigkeitsbereich
    g(a);     // g(double) aufrufen, g(int) nicht im Gültigkeitsbereich
    // ...
}
```

```
void g(int);
void g3(int);

int x = ff(5);
```

Von **`ff(5)`** wird hier **`g(double)`** aufgerufen. Die Definition von **`g(int)`** erscheint zu spät, um betrachtet zu werden – genau als wäre **`ff()`** kein Template oder **`g`** hätte eine Variable benannt.

26.3.3 Bindung am Punkt der Instanziierung

Der Kontext, in dem die Bedeutung eines abhängigen Namens ermittelt wird (§26.3.1), ergibt sich aus der Verwendung eines Templates für eine gegebene Menge von Argumenten. Dies ist der sogenannte *Punkt der Instanziierung* für diese Spezialisierung (§iso.14.6.4.1). Jede Verwendung eines Templates für eine gegebene Menge von Template-Argumenten definiert einen Punkt der Instanziierung. Für ein Funktions-Template befindet sich dieser Punkt im nächsten globalen oder durch einen Namespace definierten Gültigkeitsbereich, der seine Verwendung umschließt, unmittelbar nach der Deklaration, die diese Verwendung enthält. Zum Beispiel:

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);        // g wird am Punkt der Instanziierung gebunden
}
void h(int i)
{
    extern void g(double);
    f(i);
}
// Punkt der Instanziierung für f<int>
```

Der Punkt der Instanziierung für **`f<int>()`** liegt *außerhalb* von **`h()`**. Dies ist wichtig, um sicherzustellen, dass die in **`f()`** aufgerufene Funktion **`g()`** die globale Funktion **`g(int)`** und nicht die lokale Funktion **`g(double)`** ist. Ein nicht qualifizierter Name, der in einer Template-Definition verwendet wird, kann niemals an einen lokalen Namen gebunden werden. Lokale Namen zu ignorieren, ist unabdingbar, um viele makroartige Verhaltensweisen zu verhindern.

Damit rekursive Aufrufe möglich sind, liegt der Punkt der Instanziierung für ein Funktions-Template *nach* der Deklaration, die es instanziiert. Zum Beispiel:

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);                  // g wird am Punkt der Instanziierung gebunden
    if (a>1) h(T(a-1));    // h wird am Punkt der Instanziierung gebunden
}
```

```
enum Count { one=1, two, three }

void h(Count i)
{
    f(i);
}
// Punkt der Instanziierung für f<int>
```

Hier ist es notwendig, dass der Punkt der Instanziierung *nach* der Definition von **`h()`** erscheint, um den (indirekt rekursiven) Aufruf **`h(T(a-1))`** zu ermöglichen.

Für eine Template-Klasse oder einen Klassen-Member liegt der Punkt der Instanziierung unmittelbar *vor* der Deklaration, die ihre/seine Verwendung enthält.

```
template<typename T>
class Container {
    vector<T> v;          // Elemente
    // ...
public:
    void sort();          // Elemente sortieren
    // ...
};

// Punkt der Instanziierung von Container<int>
void f()
{
    Container<int> c;     // Punkt der Verwendung
    c.sort();
}
```

Hätte der Punkt der Instanziierung nach **`f()`** gelegen, würde der Aufruf **`c.sort()`** scheitern, um die Definition von **`Container<int>`** zu finden.

Wenn man Abhängigkeiten mithilfe von Template-Argumenten explizit darstellt, wird der Template-Code verständlicher und man kann sogar auf lokale Informationen zugreifen. Zum Beispiel:

```
void fff()
{
    struct S { int a,b; };
    vector<S> vs;
    // ...
}
```

Hier ist **`S`** ein lokaler Name, doch da wir ihn als explizites Argument verwenden, anstatt zu versuchen, seinen Namen in der Definition von **`vector`** zu vergraben, haben wir keine potenziell überraschenden Feinheiten zu erwarten.

Warum vermeiden wir dann nicht komplett nichtlokale Namen in Template-Definitionen? Dies würde sicherlich das technische Problem mit der Namenssuche lösen, doch wir wollen – wie bei normalen Funktions- und Klassendefinitionen – in der Lage sein, „andere Funktionen und Typen“ ungehindert in unserem Code zu verwenden. Wenn man jede Abhängigkeit in ein Argument umwandelt, führt das zu sehr chaotischem Code. Zum Beispiel:

```
template<typename T>
void print_sorted(vector<T>& v)
{
    sort(v.begin(),v.end());
    for (const auto& x : v)
        cout << x << '\n';
}

void use(vector<string>& vec)
{
    // ...
    print_sorted(vec);    // mit std::sort sortieren, dann mit std::cout ausgeben
}
```

Hier verwenden wir lediglich zwei nichtlokale Namen (**sort** und **cout**, die beide aus der Standardbibliothek stammen). Um diese zu eliminieren, müssten wir Parameter hinzufügen:

```
template<typename T, typename S>
void print_sorted(vector<T>& v, S sort, ostream& os)
{
    sort(v.begin(),v.end());
    for (const auto& x : v)
        os << x << '\n';
}

void fct(vector<string>& vec)
{
    // ...
    using Iter = decltype(vec.begin());     // Iteratortyp von vec
    print_sorted(vec,std::sort<Iter>,std::cout);
}
```

In diesem trivialen Fall ist ziemlich viel Code erforderlich, um die Abhängigkeit vom globalen Namen **cout** zu beseitigen. Wie aber **sort()** veranschaulicht hat, kann der Code durch zusätzliche Parameter im Allgemeinen wesentlich weitschweifiger werden, ohne dass er dadurch verständlicher wird.

Wären die Regeln für die Namensbindung bei Templates radikal restriktiver als die Regeln für Nicht-Template-Code, wäre es zudem eine vollkommen andere Herausforderung, Template-Code zu schreiben als Nicht-Template-Code. Templates und Nicht-Template-Code würden nicht mehr so einfach und problemlos zusammenarbeiten.

26.3.4 Mehrere Punkte der Instanziierung

Eine Template-Spezialisierung kann generiert werden

- an jedem Punkt der Instanziierung (§26.3.3),
- an jedem darauffolgenden Punkt in einer Übersetzungseinheit oder
- in einer Übersetzungseinheit, die speziell für das Generieren von Spezialisierungen erstellt wurde.

Dies spiegelt drei offensichtliche Strategien wider, nach denen eine Implementierung Spezialisierungen generieren kann:

1. Eine Spezialisierung generieren, wenn der erste Aufruf erscheint
2. Alle Spezialisierungen, die für eine Übersetzungseinheit erforderlich sind, am Ende der Übersetzungseinheit generieren
3. Alle Spezialisierungen, die für das Programm erforderlich sind, generieren, nachdem jede Übersetzungseinheit des Programms abgearbeitet wurde

Alle drei Strategien weisen Stärken und Schwächen auf und es sind auch Kombinationen dieser Strategien möglich.

Ein Template, das mehrfach mit derselben Menge von Template-Argumenten verwendet wird, hat entsprechend viele Instanziierungspunkte. Ein Programm ist unzulässig, wenn es möglich ist, zwei unterschiedliche Bedeutungen zu konstruieren, indem zwei verschiedene Instanziierungspunkte ausgewählt werden. Wenn sich also die Bindungen eines abhängigen oder eines unabhängigen Namens unterscheiden können, ist das Programm unzulässig. Zum Beispiel:

```
void f(int);                    // hier geht es mir um int-Werte

namespace N {
    class X { };
    char g(X,int);
}

template<typename T>
char ff(T t, double d)
{
    f(d);                       // f wird an f(int) gebunden
    return g(t,d);              // g könnte an g(X,int) gebunden werden
}

auto x1 = ff(N::X{},1.1);       // ff<N::X,double>; kann g an N::g(X,int) binden,
                                // wobei 1.1 zu 1 eingeschränkt wird


namespace N {                   // N erneut öffnen, um doubles zu berücksichtigen
    double g(X,double);
}

auto x2 = ff(N::X{},2.2);       // ff<N::X,double>; bindet g an N::g(X,double);
                                // die beste Übereinstimmung
```

Für **`ff()`** haben wir zwei Instanziierungspunkte. Für den ersten Aufruf könnten wir die Spezialisierung bei der Initialisierung von **`x1`** generieren und **`g(N::X,int)`** aufrufen lassen. Alternativ dazu könnten wir warten und die Spezialisierung am Ende der Übersetzungseinheit generieren, sodass **`g(N::X,double)`** aufgerufen wird. Folglich ist der Aufruf **`ff(N::X{},1.1)`** ein Fehler.

Es gilt als nachlässige Programmierung, eine überladene Funktion zwischen zwei ihrer Deklarationen aufzurufen. In einem großen Programm hätte ein Programmierer keinen Grund, dort ein Problem zu vermuten. In diesem konkreten Fall könnte ein Compiler die Mehrdeutigkeit abfangen. Ähnliche Probleme können allerdings auch in getrennten Übersetzungseinheiten auftreten und dann wird die Erkennung wesentlich schwerer (sowohl für Compiler als auch für Programmierer). Von einer Implementierung darf nicht erwartet werden, dass sie derartige Probleme abfängt.

Um überraschende Namensbindungen zu vermeiden, ist es am besten, Kontextabhängigkeiten in Templates zu begrenzen.

26.3.5 Templates und Namespaces

Wenn eine Funktion aufgerufen wird, lässt sich ihre Deklaration auch dann finden, wenn sie sich nicht im Gültigkeitsbereich befindet, sofern sie im selben Namespace wie eines ihrer Argumente deklariert ist (§14.2.4). Dies ist wichtig für Funktionen, die in Template-Definitionen aufgerufen werden, weil es sich hierbei um den Mechanismus handelt, durch den abhängige Funktionen während der Instanziierung gefunden werden. Die Bindung von abhängigen Namen erfolgt (§iso.14.6.4.2), indem

1. die Namen im Gültigkeitsbereich an dem Punkt, an dem das Template definiert wird, und
2. die Namen im Namespace eines Arguments eines abhängigen Aufrufs (§14.2.4)

betrachtet werden. Zum Beispiel:

```
namespace N {
    class A { /* ... */ };
    char f(A);
}

char f(int);

template<typename T>
char g(T t)
{
    return f(t);        // f() abhängig von T auswählen
}

char f(double);

char c1 = g(N::A());    // bewirkt, dass N::f(N::A) aufgerufen wird
char c2 = g(2);         // bewirkt, dass f(int) aufgerufen wird
char c3 = g(2.1);       // bewirkt, dass f(int) aufgerufen wird; f(double) wird
                        // nicht berücksichtigt
```

Hier ist **`f(t)`** zweifellos abhängig, sodass wir **`f`** am Punkt der Definition nicht binden können. Um eine Spezialisierung für **`g<N::A>(N::A)`** zu generieren, sucht die Implementierung im Namespace **`N`** nach **`f()`**-Funktionen und findet **`N::f(N::A)`**.

Die Funktion **`f(int)`** wird gefunden, weil sie sich im Gültigkeitsbereich am Punkt der Definition des Templates befindet. Dagegen wird **`f(double)`** nicht gefunden, weil sie sich am Punkt der Definition des Templates nicht im Gültigkeitsbereich befindet (§iso.14.6.4.1) und eine argumentabhängige Suche (§14.2.4) findet keine globale Funktion, die nur Argumente integrierter Typen übernimmt. Meiner Ansicht nach kann man das leicht vergessen.

26.3.6 Zu aggressive ADL

Die argumentabhängige Suche (Argument-Dependent Lookup, ADL) ist sehr hilfreich, um Weitschweifigkeit zu vermeiden (§14.2.4). Zum Beispiel:

```
#include <valarray>    // Hinweis: kein "using namespace std;"

valarray<double> fct(valarray<double> v1, valarray<double> v2, double d)
{
    return v1+d*v2;    // OK wegen ADL
}
```

Ohne argumentabhängige Suche würde der Additionsoperator **+** für **valarray** nicht gefunden werden. Wie es aussieht, bemerkt der Compiler, dass das erste Argument an **+** ein **valarray** ist, das in **std** definiert ist. Deshalb sucht er nach dem Operator **+** in **std** und findet ihn (in **<valarray>**).

Allerdings kann die ADL in Verbindung mit unbeschränkten Templates „zu aggressiv“ sein. Sehen Sie sich dazu folgenden Code an:

```
#include<vector>
#include<algorithm>
// ...

namespace User {
    class Customer { /* ... */ };
    using Index = std::vector<Customer*>;

    void copy(const Index&, Index&, int deep);    // tiefe oder flache Kopie,
                                                  // je nach dem Wert von deep

    void algo(Index& x, Index& y)
    {
        // ...
        copy(x,y,false);    // Fehler
    }
}
```

Es besteht die begründete Vermutung, dass der Autor von **User** für **User::algo()** den Aufruf **User::copy()** vorgesehen hat. Dieser findet allerdings nicht statt. Der Compiler stellt fest, dass **Index** eigentlich ein **vector** ist (in **std** definiert), sucht nach einer relevanten Funktion in **std** und findet in **<algorithm>**:

```
template<typename In, typename Out>
Out copy(In,In,Out);
```

Offensichtlich stellt dieses allgemeine Template eine perfekte Übereinstimmung für **copy(x,y,false)** dar. Andererseits kann die Funktion **copy()** in **User** nur mit einer Konvertierung von **bool** nach **int** aufgerufen werden. Für dieses Beispiel (wie auch für äquivalente Beispiele) ist die Auflösung durch den Compiler eine Überraschung für die meisten Programmierer und eine Quelle für sehr undurchsichtige Bugs. Vermutlich ist es als Designfehler der Sprache anzusehen, mithilfe von ADL völlig allgemeine Templates zu suchen. Immerhin erfordert **std::copy()** ein Paar Iteratoren (und nicht einfach zwei Argumente desselben Typs, wie die beiden **Index**-Argumente). Der Standard besagt das, der Code aber

nicht. Viele derartige Probleme lassen sich mithilfe von Konzepten lösen (§24.3, §24.3.2). Wenn zum Beispiel der Compiler erkannt hätte, dass **`std::copy()`** zwei Iteratoren verlangt, wäre das Ergebnis ein klar erkennbarer Fehler gewesen.

```
template<typename In, typename Out>
Out copy(In p1, In p2, Out q)
{
    static_assert(Input_iterator<In>(), "copy(): In ist kein Eingabeiterator");
    static_assert(Output_iterator<Out>(), "copy(): Out ist kein Ausgabeiterator");
    static_assert(Assignable<Value_type<Out>,Value_type<In>>(),
                                          "copy(): Werttypkonflikt");
    // ...
}
```

Noch besser wäre es gewesen, wenn der Compiler erkannt hätte, dass **`std::copy()`** nicht einmal ein gültiger Kandidat für diesen Aufruf ist, und **`User::copy()`** aufgerufen worden wäre. Zum Beispiel (§28.4):

```
template<typename In, typename Out,
        typename = Enable_if(<Input_iterator<In>()
                        && Output_iterator<Out>()
                        && Assignable<Value_type<Out>,Value_type<In>>()>>
Out copy(In p1, In p2, Out q)
{
    // ...
}
```

Unglücklicherweise sind viele derartige Templates in Bibliotheken untergebracht, die ein Benutzer nicht modifizieren kann (z. B. die Standardbibliothek).

Am besten vermeidet man völlig allgemeine (vollkommen unbeschränkte) Funktions-Templates in Headern, die auch Typdefinitionen enthalten können. Allerdings ist das schwer durchzusetzen. Falls Sie ein solches Template benötigen, lohnt es sich oftmals, es mit einer Einschränkungsüberprüfung zu schützen.

Was kann ein Benutzer tun, wenn eine Bibliothek unbeschränkte Templates enthält, die Probleme verursachen? Häufig wissen wir, aus welchem Namespace unsere Funktion kommen sollte, sodass wir ihn explizit angeben können. Zum Beispiel:

```
void User::algo(Index& x, Index& y)
{
    User::copy(x,y,false);      // OK
    // ...
    std::swap(*x[i],*x[j]);     // OK: nur std::swap wird betrachtet
}
```

Möchten wir den Namespace nicht spezifizieren, aber sicherstellen, dass eine bestimmte Version einer Funktion durch Überladen von Funktionen betrachtet wird, können wir eine **`using`**-Deklaration (§14.2.2) verwenden. Zum Beispiel:

```
template<typename Range, typename Op>
void apply(const Range& r, Op f)
{
    using std::begin;
    using std::end;
    for (auto& x : r)
        f(x);
}
```

Nun befinden sich die Standardfunktionen **begin()** und **end()** in der überladenen Menge, die von der bereichsbasierten **for**-Schleife verwendet wird (außer wenn **Range** über Member **begin()** und **end()** verfügt; §9.5.1).

26.3.7 Namen aus Basisklassen

Besitzt ein Klassen-Template eine Basisklasse, kann sie auf Namen aus dieser Basisklasse zugreifen. Wie bei anderen Namen gibt es zwei unterschiedliche Möglichkeiten:

- Die Basisklasse hängt von einem Template-Argument ab.
- Die Basisklasse hängt nicht von einem Template-Argument ab.

Der zweite Fall ist einfach und lässt sich behandeln wie Basisklassen in Klassen, die keine Templates sind. Zum Beispiel:

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public B {
public:
    void h(int);
    void f()
    {
        g(2);        // B::g(char) aufrufen
        h(2);        // X::h(int) aufrufen
    }
    // ...
};
```

Wie immer überdecken lokale Namen andere Namen, sodass **h(2)** an **X::h(int)** gebunden und **B::h(char)** niemals betrachtet wird. Analog dazu wird der Aufruf **g(2)** an **B::g(char)** gebunden, ohne irgendwelche Funktionen zu berücksichtigen, die außerhalb von **X** deklariert sind. Das heißt, die globale Funktion **g()** wird überhaupt nicht betrachtet.

Für Basisklassen, die von einem Template-Parameter abhängen, müssen wir etwas sorgfältiger sein und explizit angeben, was wir möchten. Sehen Sie sich dazu folgenden Code an:

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public T {
public:
    void f()
    {
        g(2);        // ::g(int) aufrufen
```

```
    }
    // ...
};
void h(X<B> x)
{
    x.f();
}
```

Warum ruft **`g(2)`** nicht **`B::g(char)`** auf (wie im vorherigen Beispiel)? Weil **`g(2)`** nicht vom Template-Parameter **`T`** abhängt. Die Funktion wird demzufolge am Punkt der Definition gebunden; Namen vom Template-Argument **`T`** (das hier als Basisklasse verwendet wird) sind (noch) nicht bekannt und werden deshalb nicht betrachtet. Sollen Namen aus einer abhängigen Klasse berücksichtigt werden, müssen wir die Abhängigkeit klarmachen. Dazu haben wir drei Möglichkeiten:

- einen Namen mit einem abhängigen Typ qualifizieren (z. B. **`T::g`**)
- angeben, dass der Name auf ein Objekt dieser Klasse verweist (z. B. **`this->g`**)
- den Namen mit einer **`using`**-Deklaration in den Gültigkeitsbereich bringen (z. B. **`using T::g`**).

Zum Beispiel:

```
void g(int);
void g2(int);

struct B {
    using Type = int;
    void g(char);
    void g2(char);
};

template<typename T>
class X : public T {
public:
    typename T::Type m;    // OK
    Type m2;               // Fehler (kein Type im Gültigkeitsbereich)

    using T::g2();         // T::g2() in den Gültigkeitsbereich bringen

    void f()
    {
        this->g(2);        // T::g aufrufen
        g(2);              // ::g(int) aufrufen; überrascht?
        g2(2);             // T::g2 aufrufen
    }
    // ...
};

void h(X<B> x)
{
    x.f();
}
```

Nur am Punkt der Instanziierung wissen wir, ob das für den Parameter **`T`** verwendete Argument (hier **`B`**) die erforderlichen Namen besitzt.

Man kann leicht vergessen, Namen von einer Basisklasse zu qualifizieren, und der qualifizierte Code sieht oftmals ein wenig weitschweifig und unordentlich aus. Die Alternative wäre aber, dass ein Name in einer Template-Klasse abhängig vom Template-Argument manchmal an einen Member der Basisklasse und manchmal an eine globale Entität gebunden würde. Das ist ebenfalls nicht ideal und die Sprachregel unterstützt die Faustregel, dass eine Template-Definition so eigenständig wie möglich sein sollte (§26.3).

Es kann recht lästig sein, den Zugriff auf abhängige Member der Basisklasse zu qualifizieren. Allerdings helfen explizite Qualifizierungen bei der Programmpflege, sodass sich der ursprüngliche Autor nicht über die zusätzliche Tipparbeit beschweren sollte. Häufig tritt dieses Problem auf, wenn eine ganze Klassenhierarchie in Templates überführt wird. Zum Beispiel:

```
template<typename T>
class Matrix_base {    // Speicher für Matrizen, Operationen aller Elemente
    // ...
    int size() const { return sz; }
protected:
    int sz;     // Anzahl der Elemente
    T* elem;    // Matrix-Elemente
};

template<typename T, int N>
class Matrix : public Matrix_base<T> {    // N-dimensionale Matrix
    // ...
    T* data()    // Zeiger auf Elementspeicher zurückgeben
    {
        return this->elem;
    }
};
```

Hier ist die Qualifizierung **`this->`** erforderlich.

26.4 Ratschläge

1. Lassen Sie den Compiler/die Implementierung Spezialisierungen je nach Bedarf generieren; §26.2.1.
2. Instanziieren Sie explizit, wenn Sie genaue Kontrolle über die Instanziierungsumgebung benötigen; §26.2.2.
3. Instanziieren Sie explizit, wenn Sie die zum Generieren von Spezialisierungen benötigte Zeit optimieren wollen; §26.2.2.
4. Vermeiden Sie in einer Template-Definition subtile Kontextabhängigkeiten; §26.3.
5. Die in einer Template-Definition verwendeten Namen müssen sich im Gültigkeitsbereich befinden oder über argumentabhängige Suche (ADL) auffindbar sein; §26.3, §26.3.5.
6. Halten Sie den Bindungskontext zwischen Instanziierungspunkten unverändert; §26.3.4.
7. Vermeiden Sie völlig allgemeine Templates, die durch ADL gefunden werden können; §26.3.6.

8. Verwenden Sie Konzepte und/oder `static_assert`, um die Verwendung ungeeigneter Templates zu vermeiden; §26.3.6.
9. Begrenzen Sie die Reichweite der ADL mit `using`-Deklarationen; §26.3.6.
10. Qualifizieren Sie Namen aus einer Template-Basisklasse je nach Bedarf mit `->` oder `T::`; §26.3.7.

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z