HANSER

Leseprobe

Bjarne Stroustrup

Eine Tour durch C++

Die kurze Einführung in den neuen Standard C++11

Übersetzt aus dem Englischen von Frank Langenau

ISBN (Buch): 978-3-446-43962-7

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43962-7

sowie im Buchhandel.

Inhalt

Vorwort

When you wish to instruct,
be brief.
– Cicero

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

Wie andere moderne Sprachen ist C++ groß und es gibt eine große Anzahl von Bibliotheken, die für den effizienten Einsatz dieser Sprache erforderlich sind. Dieses kompakte Buch soll einem erfahrenen Programmierer eine Vorstellung davon vermitteln, was modernes C++ ausmacht. Das Buch behandelt die wichtigsten Sprachfeatures und die Hauptkomponenten der Standardbibliothek. Es lässt sich zwar in wenigen Stunden lesen, doch gehört offenbar wesentlich mehr dazu, gutes C++ zu schreiben, als man an einem Tag lernen kann. Allerdings ist hier nicht Meisterschaft das Ziel. Vielmehr soll das Buch einen Überblick bieten, Prinzipbeispiele angeben und einem Programmierer bei den ersten Schritten helfen. Um das Thema zu beherrschen, sollten Sie sich mein Buch *The C++ Programming Language, Fourth Edition* (TC++PL4) [Stroustrup, 2013] ansehen. Letztlich ist dieses Buch eine erweiterte Version des Stoffes, der die Kapitel 2 bis 5 von TC++PL4 ausmacht, auch *Tour of C++* genannt. Um dieses Buch zu einem einigermaßen eigenständigen Werk zu machen, habe ich Erweiterungen und Verbesserungen hinzugefügt. Die Struktur dieser Tour folgt der von TC++PL4, sodass Sie ergänzenden Stoff leicht finden können. Auch die Übungen für TC++PL4 auf meiner Website (*www.stroustrup.com*) sind geeignet, um diese Tour zu unterstützen.

Es wird hier davon ausgegangen, dass Sie bereits programmiert haben. Andernfalls sollten Sie sich zuerst mit einem Lehrbuch wie zum Beispiel *Programming: Principles and Practice Using C++* [Stroustrup, 2009] befassen, bevor Sie hier fortfahren. Doch auch wenn Sie bereits programmiert haben, können die von Ihnen verwendete Sprache oder die Anwendungen, die Sie geschrieben haben, erheblich von dem hier vorgestellten C++-Stil abweichen.

Stellen Sie sich als Analogie eine kurze Stadtrundfahrt vor, etwa in Kopenhagen oder New York. In nur wenigen Stunden erhaschen Sie einen kurzen Blick auf die Hauptattraktionen, erfahren einige Hintergrundgeschichten und bekommen normalerweise einige Vorschläge, was Sie sich als Nächstes ansehen sollten. Nach einer solchen Tour kennen Sie die Stadt aber *nicht*. Und Sie verstehen auch *nicht*, was Sie alles gesehen und gehört haben. Auch werden Sie sich *nicht* in den formellen und informellen Regeln zurechtfinden, die das Leben in der Stadt bestimmen. Um eine Stadt wirklich kennenzulernen, müssen Sie in ihr leben – am besten für mehrere Jahre. Mit ein bisschen Glück allerdings haben Sie einen kleinen

Überblick gewonnen, eine Vorstellung davon, was die Stadt Besonderes zu bieten hat, und Anregungen, was für Sie interessant sein könnte. Nach der Tour kann die eigentliche Erkundung beginnen.

Diese Tour stellt die wesentlichen Sprachfeatures von C++ in dem Maße vor, wie sie Programmierstile unterstützen, beispielsweise die objektorientierte und generische Programmierung. Sie versucht nicht, eine detaillierte Sicht der Sprache Feature für Feature in der Art eines Referenzhandbuchs zu bieten. Analog dazu stellt sie die Standardbibliotheken in Form von Beispielen vor, ohne vollständig sein zu wollen. Außerdem werden keine Bibliotheken beschrieben, die über die im ISO-Standard definierten hinausgehen. Der Leser kann bei Bedarf nach unterstützendem Material recherchieren. Beispiele hierfür sind [Stroustrup, 2009] und [Stroustrup, 2013], doch im Web stehen enorme Mengen an weiterführenden Informationen (in unterschiedlicher Qualität) zur Verfügung. Wenn ich zum Beispiel eine Funktion oder Klasse der Standardbibliothek erwähne, lässt sich deren Definition leicht auffinden. Und wenn Sie sich die Dokumentationen der jeweiligen Header ansehen (die ebenfalls im Web zugänglich sind), werden Sie auch auf zahlreiche verwandte Elemente stoßen.

Diese Tour hier stellt C++ als geschlossene Einheit vor und nicht in Themen aufgegliedert. Deshalb kennzeichnet sie die Sprachfeatures auch nicht danach, ob sie in C vorhanden, Teil von C++98 oder neu in C++11 sind. Solche historischen Informationen finden Sie in Kapitel 14.

Danksagungen

Ein großer Teil des hier präsentierten Stoffes stammt aus „The C++ Programming Language“ (TC++PL4) [Stroustrup, 2013], 2015 auf Deutsch erschienen unter dem Titel „Die C++-Programmiersprache“. Ich danke deshalb allen, die zu diesem Buch beigetragen haben. Außerdem geht mein Dank an Peter Gordon, der zuerst vorgeschlagen hatte, die vier Tour-Kapitel aus TC++PL4 zu einer einigermaßen eigenständigen und konsistenten eigenen Publikation zu erweitern.

College Station, Texas *Bjarne Stroustrup*

5 Templates

Your quote here.
– B. Stroustrup

5.1 Einführung

Wer einen Vektor verwenden möchte, wird sicherlich nicht immer einen Vektor mit **double**-Typen brauchen. Ein Vektor ist ein allgemeines Konzept, das unabhängig vom Begriff einer Gleitkommazahl ist. Folglich sollte der Elementtyp eines Vektors unabhängig dargestellt werden. Ein *Template* (Vorlage) ist eine Klasse oder eine Funktion, die wir mit einem Satz von Typen oder Werten parametrisieren. Mit Templates stellen wir Konzepte dar, die man am besten als etwas sehr Allgemeines betrachtet, aus dem sich spezifische Typen und Funktionen erzeugen lassen, indem man Argumente – wie zum Beispiel den Elementtyp **double** – angibt.

5.2 Parametrisierte Typen

Unseren Vektor-Typ für **double**-Werte können wir zu einem Vektor für einen beliebigen Typ verallgemeinern, indem wir ihn zu einem Template machen und den konkreten Typ **double** durch einen Parameter ersetzen. Zum Beispiel:

```
template<typename T>
class Vector {
private:
    T* elem;    // elem zeigt auf ein Array von sz Elementen des Typs T
    int sz;
public:
    explicit Vector(int s);       // Konstruktor: Invariante einrichten,
                                  // Ressourcen belegen
    ~Vector() { delete[] elem; }  // Destruktor: Ressourcen freigeben

    // ... Kopier- und Verschiebeoperationen ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const { return sz; }
};
```

Das Präfix **template<typename T>** macht **T** zu einem Parameter der Deklaration, vor der er steht. Das ist die Version von C++ für den mathematischen Ausdruck „für alle T“ oder genauer „für alle Typen T“. Ein Typparameter lässt sich mit **class** genauso wie mit **typename** einführen und in älterem Code ist oftmals **template<class T>** als Präfix zu sehen.

Die Member-Funktionen können ähnlich definiert sein:

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0)
        throw Negative_size{};
    elem = new T[s];
    sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Mit diesen Definitionen können wir wie folgt **Vector**-Objekte definieren:

```
Vector<char> vc(200);        // Vektor mit 200 Zeichen
Vector<string> vs(17);       // Vektor mit 17 Strings
Vector<list<int>> vli(45);   // Vektor mit 45 Listen von Ganzzahlen
```

Der String **>>** in **Vector<list<int>>** schließt die verschachtelten Template-Argumente ab; es handelt sich also nicht um einen falsch gesetzten Eingabeoperator. Zwischen den beiden spitzen Klammern ist kein Leerzeichen erforderlich (wie noch in C++98).

Die **Vector**-Objekte können wir so verwenden:

```
void write(const Vector<string>& vs)        // Vektor von Strings
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Um die bereichsbasierte **for**-Schleife für unseren **Vector** zu unterstützen, müssen wir geeignete **begin()**-und **end()**-Funktionen definieren:

```
template<typename T>
T* begin(Vector<T>& x)
{
    return x.size() ? &x[0] : nullptr;    // Zeiger auf erstes Element oder nullptr
}

template<typename T>
T* end(Vector<T>& x)
{
    return begin(x)+x.size();             // Zeiger hinter das letzte Element
}
```

Damit können wir schreiben:

```
void f2(Vector<string>& vs)    // Vektor von Strings
{
    for (auto& s : vs)
        cout << s << '\n';
}
```

In ähnlicher Weise können wir Listen, Vektoren, Maps (d. h. assoziative Arrays) usw. als Templates definieren (Kapitel 9).

Da Templates zur Übersetzungszeit verarbeitet werden, bringen sie verglichen mit „handgestricktem Code“ keinen Overhead zur Laufzeit mit sich. Tatsächlich ist der Code, der für **`Vector<double>`** generiert wird, identisch mit dem Code, der für die Version von **`Vector`** aus Kapitel 4 generiert wird. Darüber hinaus ist der Code, der für **`vector<double>`** der Standardbibliothek generiert wird, wahrscheinlich besser (da mehr Aufwand in seine Implementierung geflossen ist).

Außer Typargumenten kann ein Template auch Wertargumente übernehmen. Zum Beispiel:

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexpr int size() { return N; }
    T[N];
    // ...
};
```

Der Alias (**`value_type`**) und die **`constexpr`**-Funktion sind dafür vorgesehen, dass Benutzer auf die Template-Argumente (nur lesend) zugreifen können.

Wertargumente sind in vielen Kontexten nützlich. Zum Beispiel ist es mit **`Buffer`** möglich, Puffer beliebiger Größe ohne den bei Verwendung von Freispeicher (dynamischem Speicher) entstehenden Overhead zu erzeugen:

```
Buffer<char,1024> glob;    // globaler Zeichenpuffer (statisch alloziert)

void fct()
{
    Buffer<int,10> buf;    // lokaler Ganzzahlpuffer (auf dem Stack)
    // ...
}
```

Ein Template-Wertargument muss ein konstanter Ausdruck sein.

5.3 Funktions-Templates

Templates lassen sich vielseitiger einsetzen als lediglich einen Container mit einem Elementtyp zu parametrisieren. So verwendet man sie ausgiebig für die Parametrisierung sowohl von Typen als auch von Algorithmen in der Standardbibliothek (§9.6, §10.6). Zum Beispiel können wir eine Funktion schreiben, die die Summe der Elementwerte eines beliebigen Containers berechnet:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c)
        v+=x;
    return v;
}
```

Das Template-Argument **`Value`** und das Funktionsargument **`v`** erlauben dem Aufrufer, den Typ und den Anfangswert des Akkumulators (der Variablen, in der die Summe aufläuft) festzulegen:

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
    int x = sum(vi,0);                  // Summe eines int-Vektors (ints addieren)
    double d = sum(vi,0.0);             // Summe eines int-Vektors (doubles addieren)
    double dd = sum(ld,0.0);            // Summe einer Liste von doubles
    auto z = sum(vc,complex<double>{}); // Summe eines Vektors von complex<double>
                                        // Anfangswert ist {0.0,0.0}
    }
```

Durch das Addieren von **`int`**-Werten in einem **`double`**-Typ lässt sich eine Zahl größer als die größte **`int`**-Zahl elegant verarbeiten. Beachten Sie, wie die Typen der Template-Argumente für **`sum<T,V>`** aus den Funktionsargumenten hergeleitet werden. Erfreulicherweise brauchen wir diese Typen nicht explizit zu spezifizieren.

Diese **`sum()`**-Funktion ist eine vereinfachte Version der Funktion **`accumulate()`** aus der Standardbibliothek (§12.3).

5.4 Konzepte und generische Programmierung

Wofür sind Templates vorgesehen? Oder anders ausgedrückt: Welche Programmiertechniken sind effektiv, wenn man Templates verwendet? Templates bieten:

- die Möglichkeit, Typen (wie auch Werte und Templates) als Argumente zu übergeben, ohne Informationen zu verlieren. Das bedeutet ausgezeichnete Möglichkeiten für Inlining, wovon aktuelle Implementierungen stark profitieren.
- verzögerte Typprüfung (die erst zur Instanziierungszeit erfolgt). Dies bietet Gelegenheit, Informationen aus verschiedenen Kontexten miteinander zu verweben.
- die Fähigkeit, konstante Werte als Argumente zu übergeben. Daraus ergibt sich die Möglichkeit, Berechnungen zur Übersetzungszeit anzustellen.

Mit anderen Worten bieten Templates einen leistungsfähigen Mechanismus für Berechnungen und Typmanipulationen zur Übersetzungszeit, was zu sehr kompaktem und effizientem Code führen kann. Denken Sie daran, dass Typen (Klassen) sowohl Code als auch Werte enthalten können.

Templates verwendet man in erster Linie und am häufigsten, um *generische Programmierung* zu unterstützen, d. h. Programmierung, die sich auf das Design, die Implementierung

und die Verwendung von allgemeinen Algorithmen konzentriert. Hier bedeutet „allgemein“, dass ein Algorithmus entworfen werden kann, der ein breites Spektrum von Typen akzeptiert, sofern diese die Anforderungen des Algorithmus an seine Argumente erfüllen. Das Template ist in C++ das Hauptinstrument für die generische Programmierung. Templates bieten parametrische Polymorphie (zur Übersetzungszeit).

Sehen Sie sich dazu die Funktion `sum()` von §5.3 an. Man kann sie für jede Datenstruktur aufrufen, die die Funktionen `begin()` und `end()` implementiert, damit die bereichsbasierte `for`-Anweisung funktioniert. Mit derartigen Strukturen arbeiten die Klassen `vector`, `list` und `map` der Standardbibliothek. Darüber hinaus wird der Elementtyp der Datenstruktur nur durch seine Verwendung beschränkt: Es muss sich um einen Typ handeln, den wir zum `Value`-Argument addieren können. Der Algorithmus `sum()` ist gewissermaßen in zweierlei Hinsicht generisch: in Bezug auf den Typ der für das Speichern der Elemente verwendeten Datenstruktur („Container“) und den Typ der Elemente.

Somit verlangt `sum()`, dass ihr erstes Template-Argument ein Art von Container ist und ihr zweites Template-Argument eine Art von Zahl. Derartige Anforderungen bezeichnet man als *Konzepte*. Leider lassen sich Konzepte nicht direkt in C++11 ausdrücken. Wir können lediglich sagen, dass das Template-Argument für `sum()` ein Typ sein muss. Es gibt zwar Techniken, um Konzepte zu überprüfen, und Vorschläge für eine direkte Sprachunterstützung von Konzepten ([Stroustrup, 2013], [Sutton, 2011]), doch gingen beide Themen über den Rahmen dieses Buchs hinaus.

Gute und nützliche Konzepte sind von elementarer Bedeutung. Zudem werden sie eher entdeckt als entworfen. Beispiele sind Ganzzahlen und Gleitkommazahlen (wie sie selbst in klassischem C definiert sind), allgemeinere mathematische Konzepte wie Feld und Vektorraum sowie Container. Diese repräsentieren die fundamentalen Konzepte eines Einsatzgebiets. Es kann eine Herausforderung sein, sie zu identifizieren und bis zum erforderlichen Grad für effektive generische Programmierung zu formalisieren.

Das Prinzip soll am Konzept *Regulär* veranschaulicht werden. Ein Typ ist regulär, wenn er sich ähnlich wie ein `int` oder ein `vector` verhält. Ein Objekt eines regulären Typs

- kann standardmäßig konstruiert werden,
- lässt sich mit einem Konstruktor oder einer Zuweisung kopieren (mit der üblichen Kopiersemantik, die zwei voneinander unabhängige und auf Gleichheit vergleichbare Objekte liefert),
- kann mit `==` und `!=` verglichen werden und
- ist frei von technischen Problemen infolge übermäßig cleverer Programmiertricks.

Ein `string` ist ein weiteres Beispiel für einen regulären Typ. Wie `int` ist `string` auch *geordnet*. Das bedeutet, dass sich zwei Strings mit den Operatoren `<`, `<=`, `>` und `>=` mit der geeigneten Semantik vergleichen lassen. Konzepte sind mehr als eine syntaktische Vorstellung. Sie verkörpern vielmehr die fundamentale Semantik. Zum Beispiel sollten Sie `+` nicht als Divisionsoperator definieren; dies widerspräche den Anforderungen an sinnvolle arithmetische Operationen.

5.5 Funktionsobjekte

Eine besonders nützliche Art von Template ist das *Funktionsobjekt* (auch als *Funktor* bezeichnet), mit dem man Objekte definiert, die sich wie Funktionen aufrufen lassen. Zum Beispiel:

```
template<typename T>
class Less_than {
    const T val;                    // Wert, mit dem verglichen werden soll
public:
    Less_than(const T& v) :val(v) { }
    bool operator()(const T& x) const { return x<val; }    // Operator aufrufen
};
```

Die Funktion namens **`operator()`** implementiert den „Funktionsaufrufs-“, „Aufrufs-“ oder „Anwendungs-“ Operator **`()`**.

Wir können benannte Variablen des Typs **`Less_than`** für einen bestimmten Argumenttyp definieren:

```
Less_than<int> lti {42};           // lti(i) vergleicht i
                                   // mit 42 mithilfe von < (i<42)
Less_than<string> lts {"Backus"};  // lts(s) vergleicht s
                                   // mit "Backus" mithilfe von < (s<"Backus")
```

Ein solches Objekt können wir wie eine Funktion aufrufen:

```
void fct(int n, const string & s)
{
    bool b1 = lti(n);    // true, wenn n<42
    bool b2 = lts(s);    // true, wenn s<"Backus"
    // ...
}
```

Derartige Funktionsobjekte sind als Argumente für Algorithmen gebräuchlich. Zum Beispiel können wir das Auftreten von Werten zählen, für die ein Prädikat **`true`** zurückgibt:

```
template<typename C, typename P>
int count(const C& c, P pred)
{
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

Ein *Prädikat* ist etwas, das wir aufrufen können, um **`true`** oder **`false`** zurückzugeben. Zum Beispiel:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "Anzahl der Werte kleiner als " << x
        << ": " << count(vec,Less_than<int>{x})
        << '\n';
    cout << "Anzahl der Werte kleiner als " << s
```

```
        << ": " << count(lst,Less_than<string>{s})
        << '\n';
}
```

Hier konstruiert **`Less_than<int>{x}`** ein Objekt, für das der Aufrufoperator den **`int`** namens **`x`** vergleicht, und **`Less_than<string>{s}`** konstruiert ein Objekt, das den **`string`** namens **`s`** vergleicht. Der Reiz dieser Funktionsobjekte liegt darin, dass sie den Wert, mit dem der Vergleich stattfinden soll, mit sich führen. Wir brauchen keine separate Funktion für jeden Wert (und jeden Typ) zu schreiben und wir kommen ohne hässliche globale Variablen aus, um Werte zwischenzuspeichern. Außerdem lassen sich einfache Funktionsobjekte wie etwa **`Less_than`** problemlos als Inline-Funktion realisieren, sodass ein Aufruf von **`Less_than`** weitaus effizienter ist als ein indirekter Funktionsaufruf. Die Möglichkeit, Daten mitzuführen, plus ihre Effizienz machen Funktionsobjekte besonders nützlich als Argumente in Algorithmen.

Funktionsobjekte, mit denen man die Bedeutung von wichtigen Operationen eines allgemeinen Algorithmus spezifiziert (wie zum Beispiel **`Less_than`** für **`count()`**) heißen auch *Richtlinienobjekte* (engl. policy objects).

Das Funktionsobjekt **`Less_than`** ist getrennt von seiner Verwendung zu definieren. Das mag unkomfortabel aussehen. Deshalb gibt es auch ein Konzept, um Funktionsobjekte implizit zu generieren:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "Anzahl der Werte kleiner als " << x
        << ": " << count(vec,[&](int a){ return a<x; })
        << '\n';
    cout << "Anzahl der Werte kleiner als " << s
        << ": " << count(lst,[&](const string& a){ return a<s; })
        << '\n';
}
```

Bei der Notation **`[&](int a){ return a<x; }`** handelt es sich um einen sogenannten *Lambda-Ausdruck*. Er generiert ein Funktionsobjekt exakt so wie **`Less_than<int>{x}`**. Die mit **`[&]`** bezeichnete *Erfassungsliste* (engl. capture list) gibt an, dass auf die verwendeten lokalen Namen (wie zum Beispiel **`x`**) über Referenzen zugegriffen wird. Hätten wir nur **`x`** „erfassen“ wollen, könnten wir das mit **`[&x]`** ausdrücken. Um dem generierten Objekt eine Kopie von **`x`** zu geben, schreiben wir **`[=x]`**. Des Weiteren bedeutet **`[ ]`**, nichts zu erfassen, **`[&]`**, alle als Referenz verwendeten lokalen Namen zu erfassen, und **`[=]`**, alle lokalen Variablen zu erfassen und als Wert zu verwenden.

Lambda-Ausdrücke können komfortabel und prägnant, aber auch undurchsichtig sein. Für nichttriviale Aktionen (sagen wir, mehr als einen einfachen Ausdruck) ziehe ich es vor, die Operation zu benennen, um damit ihren Zweck zu verdeutlichen und sie an mehreren Stellen in einem Programm verwendbar zu machen.

In §4.5.4 haben wir festgestellt, wie lästig es ist, zu viele Funktionen schreiben zu müssen, um Operationen mit **`vector`**-Zeigerelementen und **`unique_ptr`**-Objekten auszuführen, wie zum Beispiel **`draw_all()`** und **`rotate_all()`**. Funktionsobjekte (insbesondere Lambda-Ausdrücke) kommen uns hier zu Hilfe: Damit können wir das Durchlaufen des Containers von der Spezifikation, was mit jedem Element zu tun ist, trennen.

Zuerst brauchen wir eine Funktion, die eine Operation auf jedes Objekt anwendet, auf das die Elemente eines Containers von Zeigern zeigen:

```
template<typename C, typename Oper>
void for_all(C& c, Oper op)   // annehmen, dass C ein Container von Zeigern ist
{
    for (auto& x : c)
        op(*x);         // op() eine Referenz auf jedes Element übergeben, auf
                        // das gezeigt wird
}
```

Jetzt können wir eine Version von **`user()`** aus §4.5 schreiben, ohne einen Satz aller **`_all`**-Funktionen schreiben zu müssen:

```
void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    for_all(v,[](Shape& s){ s.draw(); });        // draw_all()
    for_all(v,[](Shape& s){ s.rotate(45); });    // rotate_all(45)
}
```

Ich übergebe eine Referenz auf **`Shape`** an einen Lambda-Ausdruck, sodass sich der Lambda-Ausdruck nicht darum kümmern muss, wie die Objekte im Container gespeichert werden. Insbesondere funktionieren diese **`for_all()`**-Aufrufe immer noch, wenn ich **`v`** in einen **`vector<Shape*>`** ändere.

5.6 Variadische Templates

Für ein Template lässt sich definieren, dass es eine beliebige Anzahl von Argumenten beliebiger Typen übernimmt. Eine derartiges Template heißt *variadisches Template*. Zum Beispiel:

```
void f() {}       // nichts tun

template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
    g(head);       // etwas mit head unternehmen
    f(tail...);    // erneut mit tail probieren
}
```

Wichtig bei der Implementierung eines variadischen Templates ist es, dass Sie das erste Argument von den übrigen trennen können, wenn Sie ihm eine Liste von Argumenten übergeben. Hier unternehmen wir etwas mit dem ersten Argument (dem **`head`**) und rufen dann rekursiv **`f()`** mit den übrigen Argumenten (dem **`tail`**) auf. Die Ellipse (**`...`**) steht für „den Rest“ einer Liste. Natürlich wird schließlich **`tail`** leer und wir brauchen eine eigene Funktion, die dies verarbeitet.

Diese Funktion **f()** können wir wie folgt aufrufen:

```
int main()
{
    cout << "erstes: ";
    f(1,2.2,"hello");

    cout << "\nzweites: ";
    f(0.2,'c',"yuck!",0,1,2);
    cout << "\n";
}
```

Dies ruft **f(1,2.2,"hello")** auf, was wiederum **f(2.2,"hello")** aufruft, was **f("hello")** aufruft, was **f()** aufruft. Was bewirkt nun der Aufruf **g(head)**? Offensichtlich wird er in einem echten Programm das tun, was wir für jedes Argument vorgesehen haben. Zum Beispiel können wir ihn veranlassen, sein Argument (hier **head**) in die Ausgabe zu schreiben:

```
template<typename T>
void g(T x)
{
    cout << x << " ";
}
```

Damit lautet die Ausgabe:

```
erstes: 1 2.2 hello
zweites: 0.2 c yuck! 0 1 2
```

Es sieht so aus, als ob **f()** eine einfache Variante von **printf()** ist, die beliebige Listen von Werten ausgibt – implementiert in drei Zeilen Code plus die umgebenden Deklarationen.

Die Stärke von variadischen Templates ist, dass sie beliebige Argumente akzeptieren, die Sie übergeben möchten. Ihre Schwäche liegt darin, dass die Typprüfung der Schnittstelle ein möglicherweise aufwendiges Template-Programm ist.

Aufgrund der Flexibilität von variadischen Templates greift die Standardbibliothek ausgiebig darauf zurück.

5.7 Alias

Überraschend oft ist es nützlich, ein Synonym für einen Typ oder ein Template einzuführen. Zum Beispiel enthält der Standard-Header **<cstddef>** eine Definition des Alias **size_t**:

```
using size_t = unsigned int;
```

Der tatsächliche Typ namens **size_t** ist implementierungsabhängig, sodass in einer anderen Implementierung **size_t** ein **unsigned long** sein kann. Mit dem Alias **size_t** ist der Programmierer in der Lage, portablen Code zu schreiben.

Für einen parametrisierten Typ ist es durchaus üblich, einen Alias für Typen in Bezug auf ihre Template-Argumente bereitzustellen. Zum Beispiel:

```
template<typename T>
class Vector {
public:
    using value_type = T;
    // ...
};
```

In der Tat stellt jeder Container der Standardbibliothek **`value_type`** als Name seines Werttyps bereit (Kapitel 9). Damit können wir Code schreiben, der für jeden Container funktioniert, der sich an diese Konvention hält. Zum Beispiel:

```
template<typename C>
using Element_type = typename C::value_type;  // Typ der Elemente von C

template<typename Container>
void algo(Container& c)
{
    Vector<Element_type<Container>> vec;      // Ergebnisse hier speichern
    // ...
}
```

Mit Aliasing lässt sich ein neues Template definieren, indem man einige oder alle Template-Argumente bindet. Zum Beispiel:

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;    // m ist ein Map<string,int>
```

5.8 Modell der Template-Kompilierung

Die für Templates bereitgestellte Typprüfung untersucht die Verwendung von Argumenten in der Template-Definition und nicht in Bezug auf eine explizite Schnittstelle (in einer Template-Deklaration). Dies bietet eine Übersetzungszeitvariante, die man auch als *Duck-Typing* bezeichnet („Wenn es wie eine Ente watschelt und wie eine Ente schnattert, dann ist es eine Ente“). Etwas technischer ausgedrückt: Wir operieren auf Werten und die Anwesenheit und Bedeutung einer Operation hängt allein von ihren Operandenwerten ab. Dies unterscheidet sich von der alternativen Sichtweise, dass Objekte über Typen verfügen, die die Anwesenheit und Bedeutung von Operationen bestimmen. Werte „leben“ in Objekten. Auf diese Weise arbeiten Objekte (z. B. Variablen) in C++, und nur Werte, die den Anforderungen eines Objekts genügen, lassen sich im Objekt unterbringen. Bei den Abläufen zur Übersetzungszeit mithilfe von Templates sind keine Objekte beteiligt, nur Werte.

Für die Praxis bedeutet das, dass sich die Definition eines Templates (und nicht nur seine Deklaration) im Gültigkeitsbereich befinden muss. Zum Beispiel enthält der Standard-Header **`<vector>`** die Definition von **`vector`**. Ein unglücklicher Nebeneffekt ist, dass sich ein

Typfehler erst sehr spät im Übersetzungsvorgang bemerkbar macht, was gegebenenfalls zu spektakulären Fehlermeldungen führt, weil der Compiler das Problem anhand von Informationen ermittelt, die von verschiedenen Stellen im Programm stammen.

5.9 Ratschläge

1. Der Stoff in diesem Kapitel entspricht in etwa dem, was die Kapitel 20 bis 29 von [Stroustrup, 2013] wesentlich detaillierter beschreiben.
2. Verwenden Sie Templates, um Algorithmen darzustellen, die sich auf viele Argumenttypen anwenden lassen; §5.1.
3. Verwenden Sie Templates, um Container darzustellen; §5.2.
4. Verwenden Sie Templates, um das Abstraktionsniveau des Codes anzuheben; §5.2.
5. Wenn Sie ein Template definieren, sollten Sie zuerst eine Nicht-Template-Version entwerfen und debuggen. Später fügen Sie Parameter hinzu, um die getestete Version zu verallgemeinern.
6. Templates sind zwar typsicher, doch findet die Überprüfung zu spät statt; §5.4.
7. Ein Template kann Argumenttypen ohne Informationsverlust übergeben.
8. Verwenden Sie Funktions-Templates, um Argumenttypen von Klassen-Templates herzuleiten; §5.3.
9. Templates bieten einen allgemeinen Mechanismus für die Programmierung zur Übersetzungszeit; §5.4.
10. Wenn Sie ein Template entwerfen, achten Sie sorgfältig auf die Konzepte (Anforderungen), die für seine Template-Argumente angenommen werden; §5.4.
11. Verwenden Sie Konzepte als Entwurfswerkzeug; §5.4.
12. Verwenden Sie Funktionsobjekte als Argumente für Algorithmen; §5.5.
13. Verwenden Sie einen Lambda-Ausdruck, wenn Sie ein einfaches Funktionsobjekt an nur einer Stelle benötigen; §5.5.
14. Eine virtuelle Funktion kann keine Template-Funktion sein.
15. Verwenden Sie Template-Alias, um die Notation zu vereinfachen und Implementierungsdetails zu verbergen; §5.7.
16. Verwenden Sie variadische Templates für eine Funktion, die eine variable Anzahl von Argumenten mit einer breiten Palette von Typen übernehmen soll; §5.6.
17. Verwenden Sie für homogene Argumentlisten keine variadischen Templates (bevorzugen Sie hierfür Initialisierungslisten); §5.6.
18. Achten Sie bei Verwendung eines Templates darauf, dass sich seine Definition (und nicht nur seine Deklaration) im Gültigkeitsbereich befindet; §5.8.
19. Templates bieten „Duck-Typing“ zur Übersetzungszeit; §5.8.
20. Es gibt keine getrennte Übersetzung von Templates: Binden Sie Template-Definitionen per `#include` in jede Übersetzungseinheit ein, die sie verwendet.

Index

B

C

G

H

L

M

N

O

P

Q

R

S

T

U

V

W

Z