Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Benutzerdefinierte Werttypen in C++

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (Dipl.-Inf.)

vorgelegt von

Fredrik Winkler

im September 2010

Erstbetreuer: Dr. Axel Schmolitzky
Zweitbetreuer: Dr. Alexander Pokahr

Inhaltsverzeichnis

1 Einführung

Programmieren und Modellieren sind zentrale Aktivitäten der Softwareentwicklung, die (insbesondere in der Lehre) unabhängig voneinander betrachtet werden sollten [Sch07a] [Sch07b]. Programmiersprachen beeinflussen jedoch unsere Sicht auf die Welt. Der Leitspruch vieler objektorientierter Programmiersprachen lautet „alles ist ein Objekt“. Als Folge werden häufig „Objekte“ in Anwendungsbereichen identifiziert, die konzeptionell gar keine Objekte sind. Solche nicht objektartigen Konzepte können aber häufig durch Objekte simuliert werden. Beispielsweise lassen sich Funktionen durch Objekte mit einer Operation `apply` simulieren. Simulationen dieser Art können zu Verwirrungen führen, weil es hierbei leicht zu Vermischungen der konzeptionellen und technischen Ebene kommt.

Die vorliegende Arbeit beschäftigt sich mit dem Konzept *Wert* und seiner Umsetzung in objektorientierten Programmiersprachen. Als Ausgangspunkt für die Diskussion von Werten dienen die primitiven Datentypen wie `boolean` oder `int`. Der Datentyp `boolean` hat eine feste Wertemenge, die nur aus zwei Elementen besteht: `true` und `false`. Diese Elemente werden weder erzeugt noch zerstört, und der Programmierer kann auch keine Änderungen an ihnen vornehmen. Operationen auf `boolean` wie `and` oder `not` verändern keine Werte, sondern bilden sie lediglich aufeinander ab. Solche Typen werden in dieser Arbeit als *Werttypen* bezeichnet.

Neben diesen vordefinierten Werttypen sind auch *benutzerdefinierte Werttypen* denkbar. Beispielsweise können in einem Anwendungsbereich zweidimensionale Punkte auftreten, die aus x- und y-Koordinaten bestehen. Punkte werden oft als veränderliche Objekte („verschiebbare Punkte“) modelliert, weil die Programmiersprache lediglich Objekttypen unterstützt. Punkte als Angaben in einem Koordinatensystem sind aber konzeptionell nicht verschiebbar. Man kann Objekte verschieben, die zu verschiedenen Zeiten an unterschiedlichen Punkten liegen, aber die Punkte an sich verändern sich niemals. Ansonsten wären beispielsweise geographische Koordinaten völlig nutzlos.

Mangels Unterstützung für Werttypen müssen Programmierer in objektorientierten Sprachen eigene Werttypen durch „Objekttypen mit Einschränkungen“ simulieren und sich dabei eigenständig um die Sicherstellung der Wertartigkeit kümmern. Die beiden Klassen `java.awt.geom.Point2D` und `java.util.Date` sind Musterbeispiele für Typen, die irrtümlicherweise als Objekttypen modelliert wurden, obwohl es sich bei Punkten und Datumsangaben eigentlich um Werte handelt. Daher müssen beim Überschreiten von Schnittstellen defensive Kopien angelegt werden [Blo08, Item 39], was lästig ist und leicht vergessen werden kann. Mittlerweile wird eingeräumt, dass Objekte vom Typ `Point2D` und `Date` eigentlich unveränderlich sein sollten, da dies etliche Vorteile hat [Blo08, Item 15].

Unveränderlichkeit ist aber nur ein Aspekt von Werten. Weiterhin sind Werte *unerzeugbar*, und die Operationen auf ihnen sind *seiteneffektfrei* und *referenziell transparent*. Diese Begriffe werden in den nächsten Kapiteln geklärt und machen die Beschäftigung

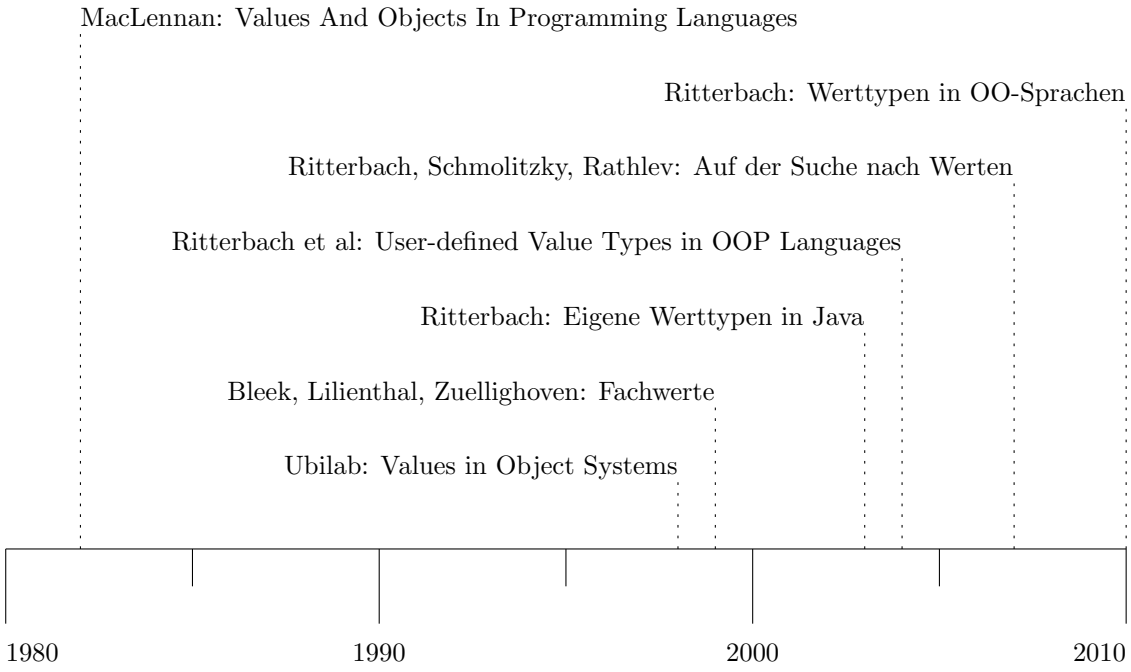

Abbildung 1.1: Bisherige Veröffentlichungen zum Thema Werttypen

mit Werttypen interessant. Dabei kann man sich zwei Fragen stellen:

1. Wie kann man Werttypen in objektorientierten Sprachen mit den vorhandenen Mechanismen simulieren?

2. Wie könnte eine Sprachunterstützung für Werttypen aussehen?

Beide Fragen sind bisher für die Sprache C++ nur sehr knapp beantwortet worden. In dieser Arbeit möchte ich im Detail untersuchen, wie benutzerdefinierte Werttypen in C++ simuliert werden können. C++ bietet einige Mechanismen an, die im Zusammenhang mit Werttypen interessant sind, welche in anderen Programmiersprachen überhaupt nicht existieren, zum Beispiel *const correctness* und die Unterscheidung von Ausdrücken in *lvalues* und *rvalues*. Diese Mechanismen sollen im Detail erläutert werden, bevor sie bei der Simulierung von Werttypen zum Einsatz kommen.

Eine Sprachunterstützung für Werttypen in C++ ist nicht Gegenstand dieser Arbeit. Ich werde jedoch kurz die Sinnhaftigkeit einer solchen Unterstützung in C++ diskutieren.

1.1 Bisherige Veröffentlichungen

Im Universitätsumfeld sind bereits etliche Veröffentlichungen zum Thema Werttypen erschienen. Die vorliegende Arbeit basiert auf den Untersuchungen von Beate Ritterbach, die zum Zeitpunkt der Verfassung dieser Arbeit ihre Dissertation über Werttypen anfertigt. Der begriffliche Grundstein wurde von MacLennan gelegt (siehe Abbildung 1.1).

Weiterhin wurden mehrere Abschlussarbeiten an der Universität Hamburg angefertigt, die sich mit der Simulierung oder Unterstützung von Werttypen befassen [Mül99] [Sau01] [FS02] [Hei05] [Rat06].

1.2 Aufbau dieser Arbeit

Kapitel 2 legt mit *Funktionen* und *Exemplaren* die begriffliche Grundlage für die vorliegende Arbeit. In Kapitel 3 wird das Sprachmodell für Werttypen vorgestellt, das von Beate Ritterbach et al. entworfen und verfeinert wurde. Die bisherigen Erkenntnisse zur Simulierung von Werttypen in Java werden in Kapitel 4 zusammengefasst.

Die im Zusammenhang mit dieser Arbeit relevanten Sprachmechanismen von C++ werden in Kapitel 5 knapp erläutert. Damit ist die in Kapitel 6 geführte Diskussion zur Umsetzung von Werttypen in C++ auch für solche Leser nachvollziehbar, die nicht mit allen Details von C++ vertraut sind. Erfahrene C++ Programmierer können Kapitel 5 weitestgehend überblättern. Lediglich die Klärung des Objektbegriffs in Abschnitt 5.2 ist obligatorisch, weil dieser stark von dem Objektbegriff des Sprachmodells abweicht.

Kapitel 7 fasst die Erkenntnisse dieser Arbeit knapp zusammen.

2 Grundlagen

In diesem Kapitel werden Grundbegriffe definiert, die für die vorliegende Arbeit zentral sind: *Funktionen* als konzeptionelle Basis für die Operationen eines Werttyps und *Exemplare* als technische Basis für die Repräsentation von zusammengesetzten Werten. Weiterhin werden *Referenzsemantik* und *Direktsemantik* voneinander abgegrenzt, weil diese bei der Diskussion von Werttypen häufig zu Verwirrungen führen.

2.1 Funktionen

In der Mathematik ist eine Funktion eine Abbildung zwischen zwei Mengen, zum Beispiel $wurzel : \mathbb{R}+ \rightarrow \mathbb{R}+$ mit $wurzel(x)^2 = x$. Aus dieser rein deklarativen Definition ist noch nicht ersichtlich, wie die Wurzel einer Zahl *berechnet* werden kann. Das Konzept einer mathematischen Funktion hat für sich genommen erst einmal nichts mit Computern zu tun. Insbesondere gibt es in der Mathematik keine Zustände oder Seiteneffekte.

In der imperativen Programmierung werden mathematische Funktionen normalerweise von *Prozeduren* berechnet (alternativ können auch Lookup-Tabellen eingesetzt werden). Beispielsweise berechnet die folgende C-Prozedur die mathematische Funktion *wurzel*:

```
double wurzel(double x)
{
    double old_guess = x;
    double guess = 1.0;

    while (guess != old_guess)
    {
        old_guess = guess;
        guess = (guess + x/guess) * 0.5;
    }
    return guess;
}
```

Diese Prozedur hat zahlreiche Schwächen im Zusammenhang mit Gleitkommazahlen, welche für die vorliegende Diskussion jedoch nicht weiter relevant sind.

2.1.1 Funktionen als eingeschränkte Prozeduren

Viele imperative Programmiersprachen behandeln *Prozedur* und *Funktion* als Synonyme, oder Funktionen sind Prozeduren, die ein Ergebnis liefern. In C++ werden Prozeduren als *functions* und Methoden als *member functions* bezeichnet.

Ich reserviere den Begriff *Funktion* in dieser Arbeit dagegen für Programmartefakte, die Funktionen im mathematischen Sinn berechnen. Um dem Konzept einer mathematischen Funktion gerecht zu werden, müssen zwei Kriterien erfüllt werden:

1. **Referenzielle Transparenz**: Das Ergebnis einer Funktion hängt ausschließlich von ihren Eingaben ab, d.h. sie liest nur Parameter und lokale Variablen aus.

2. **Seiteneffektfreiheit**: Eine Funktion hat keine nach außen sichtbaren Seiteneffekte, d.h. sie verändert nur lokale Variablen.

   Zuweisungen auf lokalen Variablen sind unproblematisch, weil sie sich nicht auf Klienten auswirken. Wenn Iteration durch Rekursion ersetzt wird, kann komplett auf Zuweisungen verzichtet werden.

Wenn man diese beiden Kriterien zusammenfasst, dann sind Funktionen vollständig vom Zustand des restlichen Programms abgeschottet. Prozeduren wie `readInput`, `writeOutput` oder `getRandomNumber` sind nach dieser Definition folglich *keine* Funktionen.

2.1.2 Vorteile von Funktionen

Der Umgang mit Funktionen hat etliche Vorteile. Das Testen von Funktionen ist zum Beispiel viel einfacher als bei anderen Prozeduren. Es ist nicht erforderlich, eine zustandsbehaftete Umgebung aufzubauen, um eine Funktion „darin“ zu testen, da das Ergebnis einer Funktion niemals von dieser Umgebung abhängen kann.

Weiterhin können beliebig viele Funktionen gleichzeitig ausgeführt werden, da diese sich nicht gegenseitig beeinflussen können. Im Zeitalter von Mehrkernprozessoren wird dieser Aspekt zunehmend interessant.

2.1.3 Sprachunterstützung für Funktionen

Ein Problem aktueller imperativer Programmiersprachen ist, dass Funktionen nicht als solche markierbar sind. Sie können lediglich mithilfe von Kommentaren und Namenskonventionen hervorgeheben werden. Das Schreiben von Funktionen ist zwar prinzipiell *möglich*, es wird jedoch nicht mit speziellen Mechanismen *unterstützt*. Eine echte Unterstützung könnte viele Programmierfehler bereits zur Übersetzungszeit abfangen:

```
// Hypothetische Programmiersprache, "function" markiert Funktion
int global_var = 42;

function int foo(int x)
{
    double y = global_var;    // error: violating referential transparency
    print(y);                 // error: calling non-functional procedure
    global_var = x;           // error: violating side-effect freeness
    return random_number();   // error: calling non-functional procedure
}
```

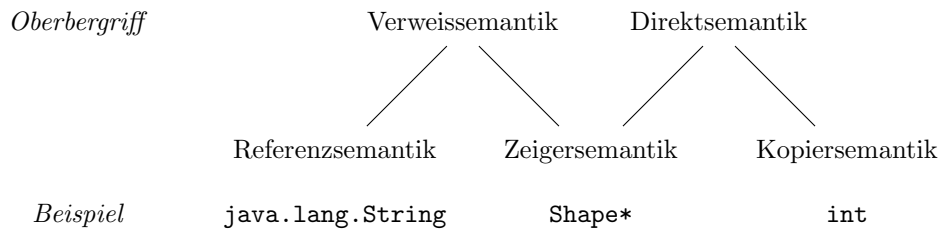

Abbildung 2.1: Variablensemantik

2.2 Exemplare

Analog zu den Funktionen und Prozeduren möchte ich bei Objekten ebenfalls zwischen *Konzept* und *Sprachmechanismus* unterscheiden. Konzeptionell hat ein Objekt einen gekapselten Zustand, der über eine wohldefinierte Schnittstelle sondiert und verändert wird. Diese Definition kann man völlig losgelöst von Programmiersprachen verstehen. Im Alltag lassen sich zahlreiche Beispiele für den Umgang mit Objekten finden.

Objektorientierte Sprachen zeichnen sich dadurch aus, dass sie das Objektkonzept mit dedizierten Sprachmechanismen direkt unterstützen. Trotzdem kann man auch in C objektorientiert programmieren [Sch94], und nicht alles, was an eine Variable vom Typ `java.lang.Object` gebunden werden kann, ist konzeptionell auch ein Objekt. Daher werde ich Objekte auf Sprachebene von nun an konsequent als „Exemplare“ bezeichnen und den Begriff „Objekt“ für die konzeptionelle Ebene reservieren.

Die ursprüngliche Motivation von Exemplaren ist die Unterstützung von Objekten. Mittlerweile werden Exemplare aber auch zur Umsetzung anderer Konzepte verwendet. Funktionen können beispielsweise nicht nur durch Prozeduren umgesetzt werden, sondern auch durch Exemplare, die eine Operation namens `apply` o.ä. anbieten. Das ist zum Beispiel bei der Parametrisierung von Sortier- und Suchalgorithmen äußerst praktisch.

Closures sind ein etabliertes Konzept aus der funktionalen Programmierung, das von einigen objektorientierten Programmiersprachen ebenfalls unterstützt wird. Unter der Haube werden Closures in der Regel durch Exemplare realisiert, die den umgebenden Kontext im Zustand eines Exemplars einfangen. Auch die in dieser Arbeit behandelten Werte können durch Exemplare realisiert werden. Trotzdem handelt es sich bei Werten um ein eigenständiges Konzept.

2.3 Variablensemantik

Falls eine Variable vom Typ `T` implizit eine Referenz auf ein `T` enthält, spreche ich von *Referenzsemantik*, andernfalls von *Direktsemantik* (siehe Abbildung 2.1). Den etablierten Begriff *Wertsemantik* vermeide ich bewusst, um Verwechslungen mit den in dieser Arbeit untersuchten Werttypen auszuschließen.

Die `class`-Typen in Java und C# werden mit Referenzsemantik behandelt, während die `struct`-Typen in C# mit Direktsemantik behandelt werden. C++ behandelt grundsätzlich alle Typen mit Direktsemantik, d.h. es gibt keine impliziten Verweise in C++.

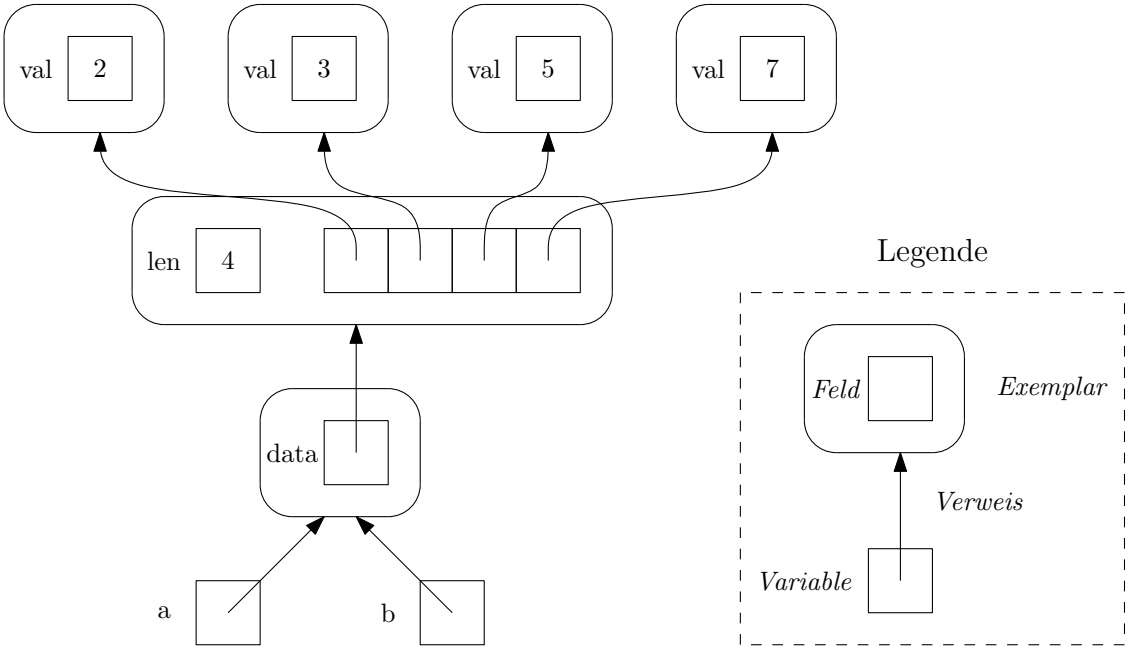

Abbildung 2.2: Referenzsemantik in Java (`java.util.Arrays.ArrayList<Integer>`)

2.3.1 Referenzsemantik

Bei zusammengesetzten Typen bewirkt Referenzsemantik, dass sich zwei Variablen nach einer Zuweisung dasselbe Exemplar teilen. Im folgenden Java-Fragment teilen sich `a` und `b` zum Beispiel dieselbe Liste, weshalb Änderungen über `b` auch über `a` sichtbar sind:

```
List<Integer> a = Arrays.asList(3, 7, 2, 5);
List<Integer> b = a;
Collections.sort(b);
System.out.println(a);      // [2, 3, 5, 7]
```

Abbildung 2.2 zeigt den Zustand nach Ablauf des Programms. `a` und `b` enthalten jeweils eine Referenz auf dasselbe `List`-Exemplar, weshalb die Sortierung über beide Variablen beobachtbar ist. Intern hält dieses Exemplar eine Referenz auf ein Array, weil Arrays in Java ebenfalls mit Referenzsemantik behandelt werden. Solange keine weiteren Referenzen auf dieses Array existieren, ist dieser Umstand jedoch irrelevant. In den Zellen dieses Arrays sind Referenzen auf `Integer`-Exemplare gespeichert. Da `Integer`-Exemplare aber unveränderlich sind, spielt es keine Rolle, wieviele Referenzen auf diese existieren.

Referenzsemantik ermöglicht außerdem das Binden einer Variable vom Typ `T` an ein Exemplar eines Subtyps von `T`, was in der objektorientierten Programmierung zentral ist und als *Subtyp-Polymorphie* bezeichnet wird. Der genaue Typ des gebundenen Exemplars ist für Klienten oft zweitrangig.

Weiterhin können Variablen bei Referenzsemantik in vielen Sprachen ungebunden sein, indem sie mit `null` belegt werden. Operationsaufrufe über ungebundene Variablen führen in sicheren Programmiersprachen zu einem Laufzeitfehler.

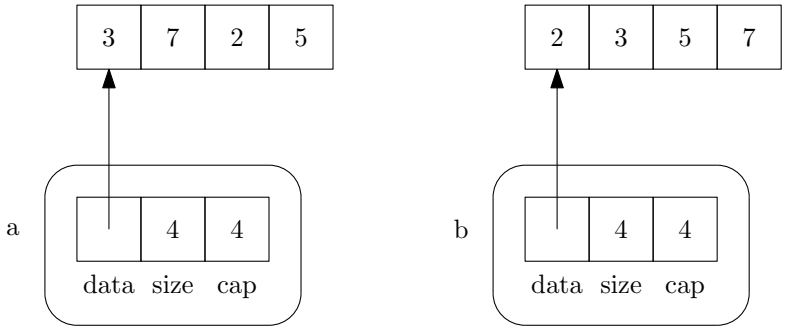

Abbildung 2.3: Kopiersemantik in C++

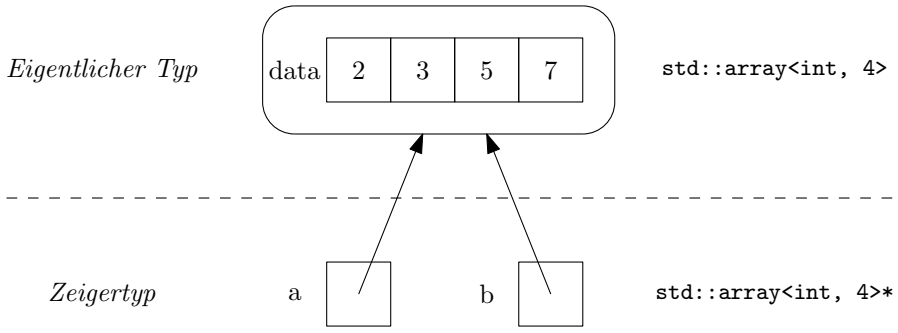

Abbildung 2.4: Zeigersemantik in C++

2.3.2 Direktsemantik

Bei Direktsemantik gibt es keine impliziten Verweise, und die Semantik der Zuweisung hängt von der verwendeten Programmiersprache und der Implementation des Typs ab.

Kopiersemantik

Wenn ein Typ mit *Kopiersemantik* behandelt wird, dann entstehen durch Zuweisungen unabhängige Kopien. Beispielsweise wird `std::vector<T>` in C++ mit Kopiersemantik behandelt. Abbildung 2.3 verdeutlicht, dass nach einer Zuweisung zwei eigenständige Vektoren existieren. Veränderungen über `a` wirken sich nicht auf `b` aus und umgekehrt.

Zeigersemantik

Referenzsemantik kann durch *Zeigersemantik* simuliert werden. Hierbei ist neben dem eigentlichen Typ noch ein zweiter Typ mit eigenen Operationen beteiligt, der Zeigertyp. Zuweisungen auf Variablen von Zeigertypen bewirken lediglich eine Kopie des Zeigers, wodurch sich beide Variablen anschließend dasselbe `T` teilen (siehe Abbildung 2.4).

Typen mit Zeigersemantik in C++ umfassen *Native Zeiger* (`T*`), *Intelligente Zeiger* (zum Beispiel `shared_ptr<T>`) und *Iteratoren* (zum Beispiel `vector<T>::iterator`).

3 Ein Modell für Werttypen in Programmiersprachen

Voraussetzung für die Diskussion von Werttypen in Programmiersprachen ist ein Verständnis des von Beate Ritterbach et al. herausgearbeiteten Wertmodells, welches in diesem Kapitel zusammengefasst wird.

3.1 Primitive Datentypen als Ausgangspunkt

Die primitiven Datentypen imperativer Programmiersprachen weisen in der Regel ein wertartiges Verhalten auf, d.h. sie erfüllen alle konzeptionellen Eigenschaften eines Werttyps. Dies möchte ich kurz am Beispiel des Datentyps `int` aus Java erläutern.

Unveränderlichkeit und Unerzeugbarkeit

Der Datentyp `int` besteht aus 4.294.967.296 verschiedenen Werten. Diese Wertemenge verändert sich nicht im Laufe der Zeit, d.h. es können keine `int`-Werte erzeugt oder zerstört werden. Jeder einzelne `int`-Wert ist ebenfalls unveränderlich.

Funktionale Natur

Die Operationen von Werten sind Funktionen im mathematischen Sinn, d.h. sie verhalten sich seiteneffektfrei und referenziell transparent (siehe Abschnitt 2.1). Der Ausdruck `2 + 3` liefert die 5 als Ergebnis, ohne irgendwelche Zustandsänderungen hervorzurufen (Seiteneffektfreiheit). Weiterhin liefert `2 + 3` bei jeder Auswertung die 5, unabhängig vom Zustand des restlichen Programms (Referenzielle Transparenz).

Seiteneffektfreiheit und Referenzielle Transparenz erlauben eine Optimierung namens *common subexpression elimination*. Beispielsweise ist es im Ausdruck `(b - a) * (b - a)` nicht erforderlich, den Teilausdruck `b - a` zweimal auszuwerten. Stattdessen kann das Ergebnis einmalig berechnet und anschließend quadriert werden. Bei Objekttypen wären solche „Optimierungen“ hingegen fatal.

3.2 Zusammengesetzte Werttypen

Es ist vorstellbar, neue Werttypen zu definieren, indem man vorhandene Werttypen zu komplexeren Werttypen zusammensetzt. Beispielsweise kann man x- und y-Koordinaten zu einem Typ `Point` zusammenfassen, welcher sich genau wie `int` wertartig verhält.

In einer hypothetischen Programmiersprache mit expliziter Unterstützung für Werttypen könnte die Definition eines Punkt-Werttyps wie folgt aussehen:

```
valueclass Point
{
    field x: int;
    field y: int;

    function add(Point p1, Point p2): Point
    {
        return select Point(p1.x + p2.x, p1.y + p2.y);
    }
}
```

(Die verwendete Syntax ist willkürlich gewählt und spielt eine untergeordnete Rolle. Zentrale Aspekte dieses Beispiels werden im folgenden erläutert.)

Bei der Herausarbeitung des Wertmodells durch Beate Ritterbach et al. war ein immer wiederkehrendes Problem, die Bestandteile eines zusammengesetzten Werttyps (im obigen Beispiel `x` und `y`) zu benennen und dabei Kollisionen mit etablierten Begriffen zu vermeiden. In Erwägung gezogene Begriffe umfassten *Feld*, *Slot*, *konstituierende Eigenschaft* und *Komponente*. In der vorliegenden Arbeit werde ich den Begriff *Feld* verwenden, auch wenn dieser Verwechslungsgefahr mit den Feldern von Objekttypen birgt.

Unveränderlichkeit

Der Typkonstruktor `valueclass` deutet an, dass `Point` eine Wertklasse ist, die sowohl einen Typ als auch dessen Implementation definiert. Im Gegensatz zu Objektklassen sind die Felder einer Wertklasse unveränderlich. Ob man die Felder öffentlich sichtbar oder durch implizit generierte Operationen zugreifbar macht, ist eine weitere von vielen Designentscheidungen, die für die vorliegende Arbeit irrelevant sind.

Unerzeugbarkeit

Wertklassen definieren keine Konstruktoren, da dies der Unerzeugbarkeit von Werten widersprechen würde. Der Punkt (1, 2) wird nicht erzeugt, sondern über eine eigene Syntax wie beispielsweise `select Point(1, 2)` *ausgewählt* (siehe obiges Beispiel). Aufgrund der referenziellen Transparenz liefern wiederholte Auswertungen von `select Point(1, 2)` immer denselben Wert, d.h. der Vergleich `select Point(1, 2) == select Point(1, 2)` ergibt immer `true`. Der Aufruf eines Konstruktors liefert dagegen jedes Mal ein neues Objekt, d.h. `new Konto(100) == new Konto(100)` ergibt grundsätzlich `false`.

Funktionale Natur

Ein effektives Mittel zur Sicherstellung von Seiteneffektfreiheit und referenzieller Transparenz ist das Benutzungsverbot von Objekttypen in Werttypen. Der Aufruf verändernder Operationen auf Objekten würde die Seiteneffektfreiheit verletzen, der Aufruf von sondierenden Operationen die referenzielle Transparenz.

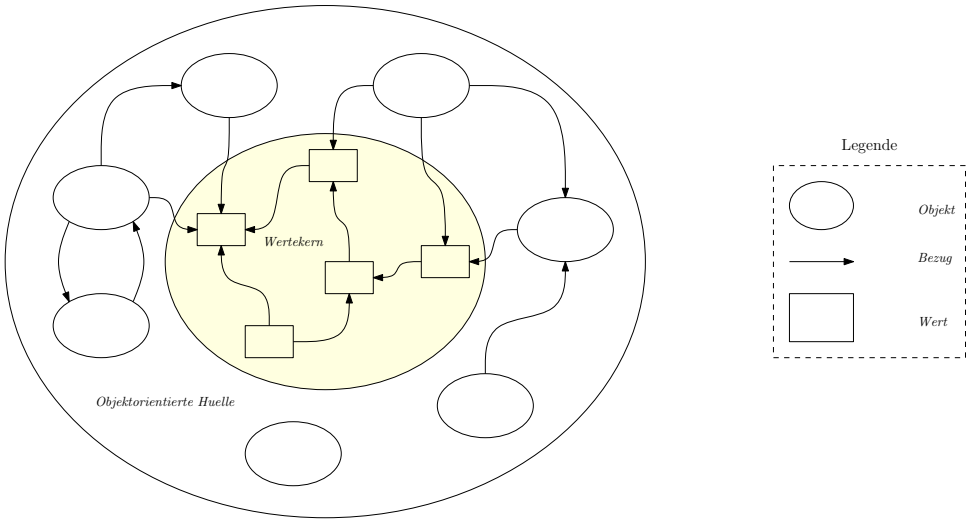

Abbildung 3.1: Wertekern und objektorientierte Hülle

Aus dieser Einschränkung ergibt sich das Bild von einem Wertekern, der von einer objektorientierten Hülle umgeben ist. Objekte können sowohl auf Objekte als auch auf Werte zugreifen, während Werte keinen Zugriff auf Objekte haben (siehe Abbildung 3.1).

Abgrenzung zu pur funktionalen Programmiersprachen

In pur funktionalen Programmiersprachen sind Variablen unveränderlich. Nach `var = exp` sind die Ausdrücke `var` und `exp` beliebig austauschbar. Dieser Umstand wird ebenfalls als *Referenzielle Transparenz* bezeichnet. Sie ermöglicht alternative Auswertungsstrategien (*lazy evaluation*) und vereinfacht die Analyse von Programmen (*equational reasoning*).

Variablen von Werttypen dürfen dagegen beliebig oft verändert werden. Es wird lediglich gefordert, dass die Werte selbst unveränderlich sind. Unveränderliche Variablen, wie sie in Java beispielsweise mittels `final` deklariert werden, sind ein orthogonales Konzept.

3.3 Sprachunterstützung

Idealerweise sollten Werttypen im Quelltext klar als solche erkennbar sein, und ein Compiler sollte die konzeptionellen Eigenschaften von Werten sicherstellen können. Dazu muss eine Programmiersprache Werttypen ausdrücklich unterstützen. Ohne eine solche Unterstützung müssen Werttypen durch Objekttypen simuliert werden. Eine solche Simulation ist fehleranfällig und führt zu Wartungsschwierigkeiten.

Die nachträgliche Erweiterung einer Programmiersprache um Werttypen hat ein grundsätzliches Problem. Werttypen dürfen sich nur auf andere Werttypen beziehen, nicht auf Objekttypen. Vor der Einführung der Spracherweiterung wurden aber vermutlich bereits zahlreiche wertartige Typen durch Objekttypen simuliert, deren Quelltexte man nicht verändern kann oder darf. Es wäre schade, wenn man diese „simulierten Werttypen“

nicht in „echten Werttypen“ benutzen könnte. Entweder muss ein Compiler überprüfen, dass es sich bei diesen simulierten Werttypen tatsächlich um Werttypen im Sinne des Modells handelt (idealerweise ohne Zugriff auf den Quelltext), oder ein Klient muss die Möglichkeit haben, beliebige Typen als Werttypen im Sinne des Modells zu kennzeichnen, was wiederum potentiell gefährlich ist.

3.4 Zusammenfassung

Im Sprachmodell, das dieser Arbeit zugrunde liegt, sind die definierenden Eigenschaften von Werten *Unerzeugbarkeit*, *Unveränderlichkeit*, *Seiteneffektfreiheit* und *Referenzielle Transparenz*. Im Gegensatz zu Objekten werden Werte nicht durch Konstruktoren erzeugt, sondern durch Selektoren ausgewählt. Innerhalb von Wertklassen ist kein Zugriff auf Objekte erlaubt, um die funktionale Natur der Wertoperationen sicherzustellen.

Eine dedizierte Sprachunterstützung für Werttypen, welche die konzeptionellen Eigenschaften von Werten sicherstellt, erscheint sinnvoll.

4 Werttypen in Java

Nachdem das Wertmodell im letzten Kapitel vorgestellt wurde, stelle ich im folgenden dessen Umsetzung in Java dar. Java ist eine objektorientierte Programmiersprache. Der Typkonstruktor `class` erlaubt die Definition eigener Objektklassen, für Wertklassen gibt es dagegen keine explizite Unterstützung. In diesem Kapitel untersuche ich kurz einige eingebaute Werttypen in Java. Dann zeige ich, wie man Wertklassen durch Objektklassen simulieren kann und diskutiere die Nachteile einer solchen Simulation. Anschließend stelle ich kurz am Beispiel VJ dar, wie eine echte Sprachunterstützung aussehen kann.

4.1 Eingebaute Werttypen

4.1.1 Primitive Werttypen

Javas primitive Datentypen `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` und `double` sind Werttypen im Sinne dieser Arbeit. Sie haben fest definierte Wertemengen, aus denen mittels Literalen unveränderliche Werte ausgewählt werden. Die Operationen verhalten sich seiteneffektfrei und referenziell transparent.

4.1.2 Zeichenketten

Zeichenketten werden in Java durch die Klasse `java.lang.String` realisiert und können weitgehend als Werttypen angesehen werden. `String`-Exemplare sind unveränderlich, und an der Schnittstelle tauchen keine Objekttypen auf. Alle Operationen sind seiteneffektfrei und referenziell transparent.

Gleiche Stringliterale bezeichnen dasselbe `String`-Exemplar [GJSB05, 3.10.5]. Zur Laufzeit können aber beliebig viele `String`-Exemplare dieselbe Zeichenkette repräsentieren. Beim Vergleich mittels `x == y` werden lediglich Referenzen verglichen, was nicht immer das gewünschte Ergebnis liefert:

```
String a = "hello".toUpperCase();
String b = "hello".toUpperCase();
System.out.println(a == b);   // false

String c = a.toUpperCase();
System.out.println(a == c);   // true
```

Falls die Eingabe vollständig aus Großbuchstaben besteht, liefert `toUppercase` einfach `this` zurück, weshalb `a == c` true liefert. Demgegenüber liefert `"hello".toUpperCase()` bei jeder Auswertung ein neues Exemplar, weshalb `a == b` false liefert.

Klienten müssen eigenständig darauf achten, Strings mit `x.equals(y)` zu vergleichen. Es ist bedauerlich, dass die syntaktisch lesbarere Form `x == y` bei Strings sinnlos ist. Eine entsprechende Sonderbehandlung in der Sprache wäre hier konsequent gewesen.

4.1.3 Wrapperklassen

Die generischen Sammlungen aus dem *Java Collections Framework* können ausschließlich mit Referenztypen parametrisiert werden. Um primitive Werte in diesen Sammlungen benutzen zu können, müssen diese durch sog. *Wrapperklassen* verpackt werden.

Die impliziten Konvertierungen zwischen primitiven Typen und Wrappertypen heißen *Auto-Boxing* und *Auto-Unboxing*. Auto-Boxing liefert für kleine Zahlen immer dasselbe Wrapper-Exemplar und für große Zahlen immer ein neues. (Die Grenzen sind nicht scharf spezifiziert.) Sofern man kein `new` verwendet, werden kleine Zahlen korrekt behandelt:

```
Integer a = 127;                // Auto-Boxing
Integer b = 127;                // Auto-Boxing (selbes Exemplar)
Integer c = new Integer(127);
System.out.println(a == b);     // Referenzvergleich -> true
System.out.println(a == c);     // Referenzvergleich -> false
System.out.println(a == 127);   // Wertvergleich -> true
```

Bei größeren Zahlen ergeben sich auch ohne explizite `new`-Aufrufe Probleme:

```
Integer x = 128;                // Auto-Boxing
Integer y = 128;                // Auto-Boxing (neues Exemplar)
Integer z = new Integer(128);
System.out.println(x == y);     // Referenzvergleich -> false
System.out.println(x == z);     // Referenzvergleich -> false
System.out.println(x == 128);   // Wertvergleich -> true
```

Genau wie bei `String` müssen Klienten bei den Wrapperklassen also im Allgemeinen eigenständig darauf achten, `x.equals(y)` statt `x == y` zu schreiben.

Selbst bei `Boolean` ist es möglich, mehrere Exemplare zu erzeugen, die denselben Wert repräsentieren. Dabei sind die Konstanten `Boolean.TRUE` und `Boolean.FALSE` sowie die statische `valueOf`-Operation aus Klientensicht völlig ausreichend. Hier wäre es sinnvoll gewesen, die `Boolean`-Konstruktoren zu verbergen, um zumindest bei `Boolean`s auf `equals` verzichten zu können.

4.2 Benutzerdefinierte Werttypen simulieren

Die Simulation von Werttypen durch Objekttypen in Java erfordert Disziplin vom Programmierer, um die geforderten Eigenschaften von Werttypen sicherzustellen.

4.2.1 Aufzählungswerttypen

Seit Java 5 gibt es den Typkonstruktor `enum` für benutzerdefinierte Aufzählungstypen. Technisch ist ein `enum E` als gewöhnliche Klasse realisiert, die von `java.lang.Enum<E>` erbt. Die deklarierten Namen sind öffentlich sichtbare Klassenkonstanten, in denen Referenzen

auf die Exemplare dieser Klasse gespeichert sind [Blo08, Item 30]. Ein `enum` ist also nichts weiter als eine Klasse, von der es zur Laufzeit eine festgelegte Anzahl von Exemplaren gibt. Bei einem einzigen Exemplar handelt es sich um ein *Singleton* [Blo08, Item 3].

In ihrer einfachsten Form können `enums` als Werttypen angesehen werden. Dabei werden lediglich die Werte aufgezählt:

```
enum Wochentag
{
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG;
}
```

Da hierbei jeder Wert durch genau ein Exemplar repräsentiert wird, ist der Vergleich mittels `x == y` korrekt. Es ist auch möglich, die Exemplare eines Aufzählungstyps mit einem Zustand zu versehen und Operationen zu implementieren:

```
enum Monat
{
    JANUAR(31), FEBRUAR(29), MÄRZ(31), APRIL(30), MAI(31), JUNI(30), JULI(31),
    AUGUST(31), SEPTEMBER(30), OKTOBER(31), NOVEMBER(30), DEZEMBER(31);

    private Monat(int tage)
    {
        _tage = tage;
    }

    public int tage()
    {
        return _tage;
    }

    private final int _tage;
}
```

Auch in diesem konkreten Beispiel wird ein Werttyp definiert. Allgemein können Unveränderlichkeit, Seiteneffektfreiheit und referenzielle Transparenz bei `enums` jedoch nicht garantiert werden.

4.2.2 Zusammengesetzte Werttypen

Da Java keinen dedizierten Typkonstruktor für zusammengesetzte Werttypen anbietet, müssen Wertklassen mit Objektklassen simuliert werden. Dabei sind gewisse Regeln zu befolgen, die ich im folgenden zusammenfasse.

Unveränderlichkeit

Die Felder einer Wertklasse sollten mittels `final` vor versehentlichen Änderungen geschützt werden. Diese Markierung erzwingt außerdem eine explizite Initialisierung der Felder durch den Programmierer. Darüber hinaus garantiert `final`, dass alle beteiligten Threads dieselben Belegungen sehen [GJSB05, 17.5].

Unerzeugbarkeit

Es empfiehlt sich, die Konstruktoren als `private` zu deklarieren und durch statische Auswahl-Methoden namens `select` zu ersetzen. Das Verstecken der Konstruktoren verhindert auch das versehentliche Bilden eines Subtyps, der sich nicht an die Spielregeln hält. Pragmatisch kann man den Konstruktor auch öffentlich lassen, dann sollte aber die Klasse mit `final` markiert werden, um Subtyping zu verhindern.

Die Vergleichsoperation

Für eine korrekte Semantik des Vergleichs gibt es drei Vorgehensweisen:

1. Zu jedem Wert gibt es *genau ein* Exemplar, das diesen Wert repräsentiert, aber die Wertemenge ist zu groß, um sie mit einem `enum` aufzuzählen:

```
class TagImJahr
{
    private final Monat _monat;
    private final int _tag;

    private TagImJahr(Monat monat, int tag)
    {
        _monat = monat;
        _tag = tag;
    }

    public static TagImJahr select(Monat monat, int tag)
    {
        if ((tag < 1) || (tag > monat.tage()))
            throw new IllegalArgumentException();
        return pool.get(monat)[tag];
    }

    private static final Map<Monat, TagImJahr[]> pool;

    static
    {
        pool = new EnumMap<Monat, TagImJahr[]>(Monat.class);
        for (Monat monat: Monat.values())
        {
            int tage = monat.tage();
            TagImJahr[] array = new TagImJahr[tage + 1];
            for (int tag = 1; tag <= tage; ++tag)
            {
                array[tag] = new TagImJahr(monat, tag);
            }
            pool.put(monat, array);
        }
    }
}
```

Alternativ kann der Pool auch nach Bedarf befüllt werden, aber dann ist es teurer, die Wertklasse *thread-safe* zu machen.

2. Zu jedem Wert gibt es *höchstens ein* Exemplar. Die Exemplare werden durch einen dynamischen Pool verwaltet, der nach Bedarf wächst und schrumpft, weil die Wertemenge zu groß ist, um für jeden Wert ein Exemplar im Speicher zu halten. Auf modernen VMs ist Pooling allerdings nicht mehr zu empfehlen [Blo08, S. 23].

3. Zu jedem Wert gibt es *beliebig viele* repräsentierende Exemplare. Dann müssen `equals` und `hashCode` überschrieben werden, und ein Klient muss `x.equals(y)` statt `x == y` verwenden. Der Konstruktor kann aus pragmatischen Gründen `public` sein, aber dann muss die Klasse mit `final` markiert werden, um Subtyping zu verhindern:

```
final class Datum
{
    private final int _jahr;
    private final TagImJahr _tagImJahr;

    public Datum(int jahr, TagImJahr tagImJahr)
    {
        _jahr = jahr;
        _tagImJahr = tagImJahr;
    }

    public boolean equals(Object o)
    {
        boolean result = false;
        if (o instanceof Datum)
        {
            Datum d = (Datum) o;
            result = (_jahr == d._jahr) && (_tagImJahr == d._tagImJahr);
        }
        return result;
    }

    public int hashCode()
    {
        return _tagImJahr.hashCode() + _jahr;
    }
}
```

Werte kennen keine Objekte

Der Programmierer muss eigenständig darauf achten, dass für die Felder und in der Schnittstelle eines Werttyps ausschließlich Werttypen verwendet werden. In den Methoden kann die Verwendung von lokal erzeugten Objekten aus pragmatischen Gründen zugelassen werden, falls die Spielregeln dadurch nicht verletzt werden. Ein prominentes Beispiel ist das Zusammenbauen einer längeren Zeichenkette durch einen `StringBuilder`.

4.3 Sprachunterstützung

Die bisherige Diskussion hat gezeigt, dass die eigenständige Sicherstellung der Wertartigkeit durch den Programmierer nicht trivial und damit fehleranfällig ist. Daher scheint es sinnvoll, eine echte Sprachunterstützung für Werttypen zu haben. Mit einer solchen Unterstützung könnte ein Compiler die konzeptionellen Eigenschaften von Werttypen automatisch sicherstellen.

Jörg Rathlev hat 2006 in seiner Diplomarbeit eine Spracherweiterung namens VJ (als Akronym für *ValueJava*) entworfen und implementiert, die einen eigenen Typkonstruktor für Werttypen anbietet [Rat06]. Das `Point`-Beispiel sieht in VJ wie folgt aus:

```
valueclass Point
{
    public int x;
    public int y;

    public Point add(Point other)
    {
        return const Point(x + other.x, y + other.y);
    }
}
```

Die Unerzeugbarkeit von Werten wird in VJ durch *Wertwähler* realisiert, für deren Aufruf das reservierte Wort `const` verwendet wird. Die Felder von Wertklassen sind in VJ grundsätzlich unveränderlich. Um die Seiteneffektfreiheit und Referenzielle Transparenz zu gewährleisten, verbietet VJ in Werttypen die Verwendung von Objekttypen. Der Infix-Test `a == b` wird bei Werttypen auf `a.equals(b)` abgebildet. Teilweise sind dazu Typprüfungen zur Laufzeit notwendig.

Nach Abschluss von Jörg Rathlevs Diplomarbeit übernahm Beate Ritterbach die Entwicklung des VJ-Compilers. 2008 realisierte Marek Walczak ein Eclipse-Plugin für VJ, um die Zugänglichkeit der Sprache zu erhöhen [Wal08].

4.4 Zusammenfassung

Die Unterstützung von benutzerdefinierten Werttypen ist in Java eher schwach. Sofern eine Wertklasse nicht durchgehend Pooling verwendet, müssen Variablen auf Klientenseite mit `equals` verglichen werden. Ein Vergleich mit `==` ist übersetzbar, aber sinnlos.

Der Programmierer der Wertklasse muss `equals` und `hashCode` manuell überschreiben oder Pooling-Mechanismen einsetzen. Die Unveränderlichkeit von Wertexemplaren kann mit konstanten Feldern sichergestellt werden, während es für die Seiteneffektfreiheit und Referenzielle Transparenz keine Unterstützung gibt. Das Verbergen der Konstruktoren muss ebenfalls manuell geschehen, weshalb dieser Aspekt leicht vergessen werden kann.

In diesem Kapitel wurde am Beispiel Java gezeigt, dass man benutzerdefinierte Werttypen prinzipiell mit bestehenden Sprachmechanismen objektorientierter Programmiersprachen simulieren kann. Wünschenswert ist jedoch eine echte Sprachunterstützung für Werttypen, um deren konzeptionelle Eigenschaften garantieren zu können.

5 C++ Fundament

In diesem Kapitel werden die grundlegenden Sprachmechanismen von C++ geklärt, die für die Untersuchung von Werttypen in C++ relevant sind.

5.1 Ursprung und Entwicklung

1979 entwarf Bjarne Stroustrup die Sprache *C with classes*, um die Eleganz von Simula mit der Effizienz von C zu verbinden. 1983 wurde die Sprache um virtuelle Methoden ergänzt und in *C++* umbenannt. Zwei Jahre später folgte das erste offizielle Release.

Mit der Einführung von Mehrfachvererbung im Jahr 1989 war die Unterstützung objektorientierter Programmierung im Wesentlichen abgeschlossen. Spätere Erweiterungen konzentrierten sich größtenteils auf ein anderes Paradigma, das unter der Bezeichnung *Generische Programmierung* bekannt ist (siehe Abbildung 5.1).

5.1.1 Philosophie

C++ wurde mit dem Anspruch entwickelt, C als Systemprogrammiersprache abzulösen [Str94, 4.5]. Daher ist C++ im Kern genauso effizient und unsicher wie C. Aufbauend auf diesem unsicheren Kern bietet C++ weitere Sprachmechanismen, die es erlauben, einfach verwendbare und sichere Bibliotheken zu schreiben.

Anwendungsprogrammierer müssen sich um Details wie Speicherverwaltung, Zeiger oder Arrays immer seltener Gedanken machen, da es mittlerweile zahlreiche Bibliotheken auf einem höheren Abstraktionsniveau gibt. Dieser Trend spiegelt sich auch in den aktuellen C++ Lehrbüchern wider, zum Beispiel [KM00], [LML05] und [Str08].

5.1.2 Standardisierung

Die Sprache wird von den Komitees ANSI-X3J16 und ISO-WG-21 standardisiert. Bisher sind zwei offizielle Standards erschienen (ISO/IEC 14882:1998 und ISO/IEC 14882:2003), wobei der zweite Standard keine wesentlichen Neuerungen mit sich bringt, sondern hauptsächlich Fehler und Unschärfen in der Formulierung beseitigt. Ein integraler Bestandteil des ersten Standards ist die *Standard Template Library*, die eine Vielzahl von generischen Sammlungen und Algorithmen umfasst [Jos99].

Geplante Bibliothekserweiterungen werden in *Technical Reports* beschrieben, bevor sie in den nächsten Sprachstandard einfließen [Bec06]. Implementierungen dieser Technical Reports findet man in der *Boost Library* [Kar05], mittlerweile werden sie aber auch von einigen Compiler-Herstellern direkt mitgeliefert.

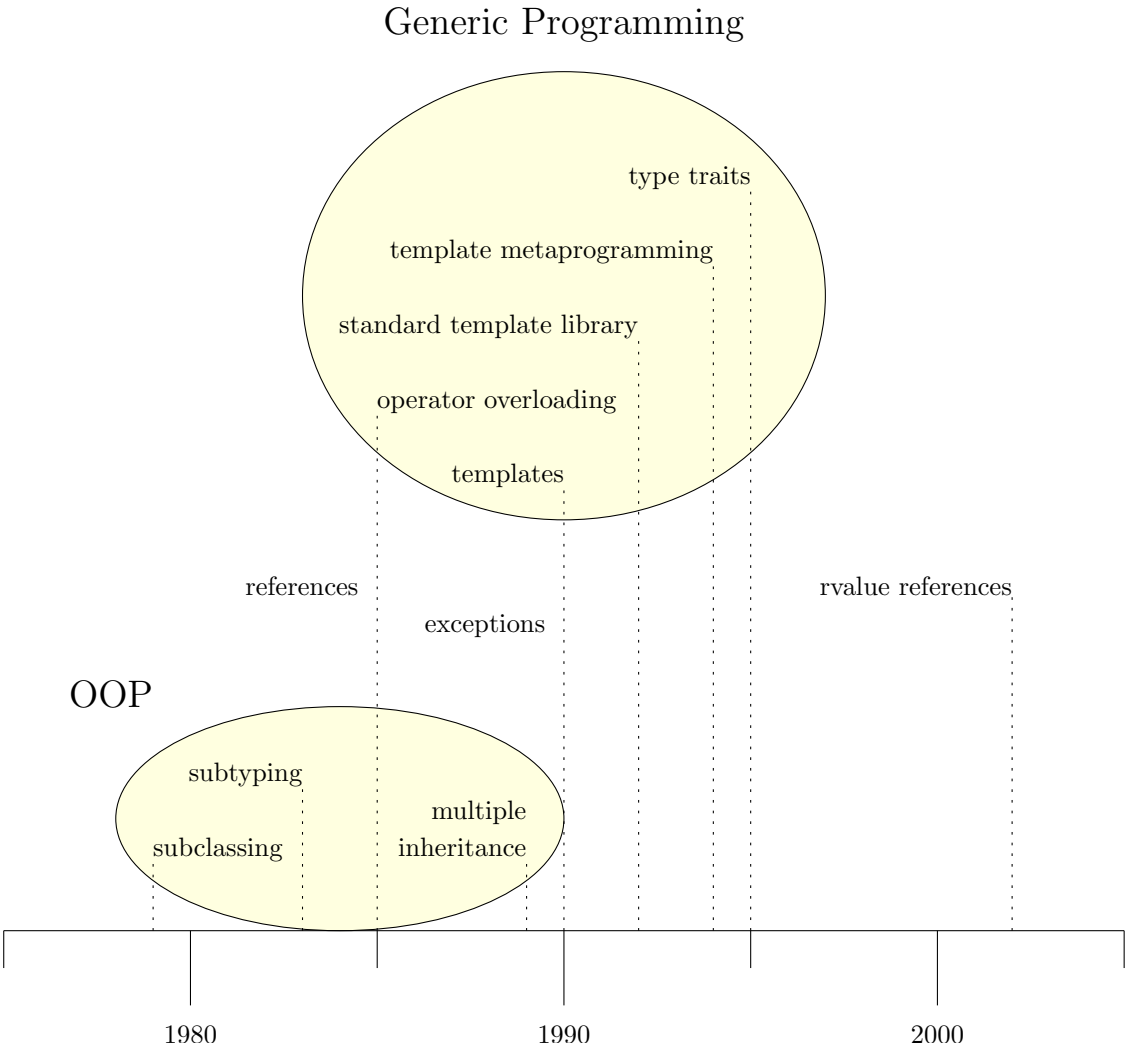

Abbildung 5.1: Entwicklung objektorientierter und generischer Programmierung in C++

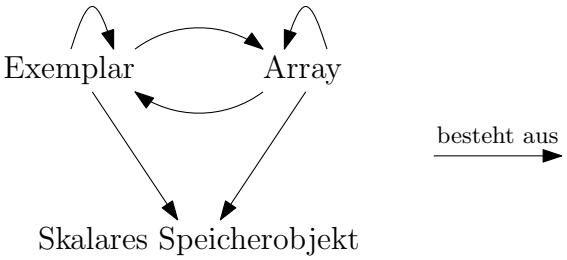

Abbildung 5.2: Speicherobjekte und ihre Beziehungen untereinander

5.2 Der ungewöhnliche Objektbegriff von C++

Als speichernahe Sprache hat C++ einen ungewöhnlichen Objektbegriff:

> The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage. [...] An object has a *type*. [ISO03, 1.8 intro.object §1]

Um Verwechslungen mit dem Objektkonzept aus Kapitel 2.2 auszuschließen, werde ich C++ Objekte von nun an konsequent als *Speicherobjekte* (*storage objects*) bezeichnen. Abbildung 5.2 zeigt alle Ausprägungen von Speicherobjekten: *Exemplare*, *Arrays* und *skalare Speicherobjekte*. Skalare Typen umfassen die arithmetischen Typen, Aufzählungstypen und Zeigertypen. Die oberen Pfeile verdeutlichen die Direktsemantik von C++: Exemplare und Arrays können andere Exemplare und Arrays direkt enthalten.

Weiterhin möchte ich diese Speicherobjekte explizit von den Speicherbereichen unterscheiden, in welche sie eingebettet sind. *Speicherbereiche* sind nichts weiter als typlose Sequenzen von Bytes im Speicher. Tabelle 5.1 zeigt zentrale C++ Begriffe sowie deren Übersetzungen, die im folgenden verwendet werden.

C++ Begriff	**Übersetzung**
region of storage	Speicherbereich (typlos)
object	Speicherobjekt (getypt)
scalar object	skalares Speicherobjekt
array	Array
array element	Zelle
class object	Exemplar
polymorphic object	Exemplar
function	Prozedur
member function	Methode (Implementation) bzw. Operation (Schnittstelle)
data member	Feld
subobject	Komponente (Oberbegriff für Feld und Zelle)
complete object	Komplettes Speicherobjekt (Gegenteil von Komponente)

Tabelle 5.1: Übersetzung zentraler C++ Begriffe

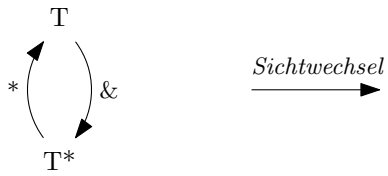

Abbildung 5.3: Zeiger im Typsystem

5.3 Relevante Eigenschaften des Typsystems

5.3.1 Zeiger

Jedes Speicherobjekt hat eine Adresse, an der es im Speicher abgelegt ist. Diese kann man in C++ über den *Adressoperator* `&` ermitteln:

```
int i = 1025;
std::cout <<  i << std::endl;   // 1025
std::cout << &i << std::endl;   // 0x22ff6c
```

Die Adresse `0x22ff6c` als solche hat keine tiefere Bedeutung. Man beachte, dass der Adressoperator bei skalaren Typen nur auf solche Ausdrücke angewendet werden kann, deren Ergebnis eine skalare Variable ist. Überraschenderweise ist der Adressoperator auch bei Exemplaren nur eingeschränkt anwendbar (Details folgen in Abschnitt 6.3.2).

Zeiger sind getypte Adressen

Die Ausgabe des Programms erweckt den Anschein, als ob der Adressoperator lediglich die Adresse `0x22ff6c` liefern würde. Darüber hinaus konserviert er aber auch die Typinformation, dass an dieser Adresse eine `int`-Variable abgelegt ist. Derart getypte Adressen heißen *Zeiger*. Der Typ des Ausdrucks `&i` ist beispielsweise `int*` („Zeiger auf `int`“).

Eine Adresse gibt lediglich Auskunft darüber, wo ein Speicherbereich anfängt. Aufgrund von statischen Typinformationen weiß ein Zeiger darüber hinaus, wie groß dieser Speicherbereich ist, und wie er zu interpretieren ist. Eigentlich müsste der Adressoperator *Zeigeroperator* heißen, weil er keine Adresse liefert, sondern einen Zeiger [ISO03, 5.3.1 expr.unary.op §2].

Dereferenzierung

Das Gegenstück zum Adressoperator ist der *Dereferenzierungsoperator* `*`. Er liefert als Ergebnis das Speicherobjekt, auf das der Zeiger zeigt. Folglich ist `*(&i)` die `int`-Variable an der Adresse von `i`, also `i`. So gesehen wechseln die Operatoren `&` und `*` zwischen zwei verschiedenen Sichten auf dasselbe Speicherobjekt (siehe Abbildung 5.3):

- direkt: das Speicherobjekt `x` vom Typ `T`, das an der Adresse `p` abgelegt ist
- indirekt: die Adresse `p`, an der das Speicherobjekt `x` vom Typ `T` abgelegt ist

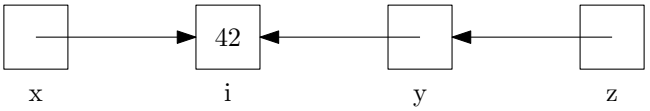

Abbildung 5.4: Zeigervariablen

Zeigervariablen

Zeiger können in passend getypten *Zeigervariablen* abgelegt werden. Abbildung 5.4 veranschaulicht das folgende Quelltextfragment:

```
int   i = 42;
int*  x = &i;
int*  y =  x;
int** z = &y;
```

Verschiedene Zeigervariablen können denselben Zeiger enthalten (`x` und `y` zeigen beide auf `i`). Außerdem können Zeigervariablen genau wie alle anderen Variablen referenziert werden (`z` zeigt auf `y`). Die Ausdrücke `*x`, `*y` und `**z` sind Aliase für die Variable `i`. Eine Dereferenzierung mittels `*` entspricht dem Verfolgen eines Pfeils in Abbildung 5.4.

Dynamische Speicherobjekte

Dynamisch erzeugte Speicherobjekte müssen prinzipiell indirekt angesprochen werden, da sie keine Namen haben. Die Erzeugung erfolgt mittels `p = new T(args)`. Mangels Garbage Collection muss die Zerstörung explizit mittels `delete p` angefordert werden:

```
int*  x = new int(42);
int*  y =  x;
int** z = &y;
// ...
delete x;
```

Dabei zerstört `delete p` das von `p` referenzierte Speicherobjekt `*p` und nicht etwa die Zeigervariable `p` selbst. Letztere kann anschließend durchaus denselben Zeiger enthalten wie vorher. Zugriffsversuche auf `*p` führen dann zu undefiniertem Verhalten.

void*

Der einzige ungetypte Zeiger `void*` kann die Adresse aller Speicherobjekte aufnehmen. Mangels Typinformation ist `void*` jedoch nicht dereferenzierbar (es gibt keine Speicherobjekte vom Typ `void`). Jeder getypte Zeiger kann implizit in `void*` konvertiert werden.

`void*` spielt eine zentrale Rolle bei der Verwaltung von (typlosen) Speicherbereichen, was die folgenden Signaturen aus der C-Standardbibliothek beispielhaft verdeutlichen:

```
void* malloc(size_t number_of_bytes);
void* calloc(size_t number_of_elements, size_t bytes_per_element);
void* realloc(void* address, size_t new_size);
void  free(void* address);
```

Abbildung 5.5: Arrays im Typsystem

5.3.2 Arrays

Ein Array vom Typ `T[n]` ist eine Sequenz von `n` anonymen Speicherobjekten des Typs `T`, die direkt hintereinander im Speicher abgelegt sind. Der Zugriff auf die `i`-te Zelle eines Arrays namens `a` erfolgt über die Notation `a[i]`. Gültige Indizes reichen von `0` bis `n-1`. Ein Zugriff außerhalb dieser Grenzen führt zu undefiniertem Verhalten. In der Praxis treten dabei schwierig zu lokalisierende Fehler auf.

Um layoutkompatibel mit C zu bleiben, legt C++ keinerlei Metainformationen in Arrays ab. Daher können ungültige Zugriffe zur Laufzeit nicht als solche erkannt werden. Eine sichere Alternative zu `T[n]` ist `std::vector<T>`. Vektoren kennen ihre eigene Größe, wachsen dynamisch und können bei Bedarf Bereichsüberprüfungen vornehmen.

Dynamische Arrays

Ein dynamisches Array wird mittels `p = new T[n]` erzeugt, wobei `p` auf die erste Zelle des Arrays zeigt. Die Zerstörung eines dynamisch erzeugten Array muss mittels `delete[] p` erfolgen. Ein Auslassen der eckigen Klammern führt zu undefiniertem Verhalten.

Dynamisch erzeugte Arrays werden häufig eingesetzt, wenn die Größe erst zur Laufzeit feststeht oder sogar dynamisch variiert. `std::vector<T>` ist hier eigentlich immer die bessere Wahl.

Arrays und Zeiger

Jedes Array vom Typ `T[n]` kann unabhängig von seiner Länge `n` implizit in einen Zeiger vom Typ `T*` konvertiert werden, wobei die Längeninformation `n` verlorengeht [ISO03, 4.2 conv.array §1]. Das Ergebnis dieser Konvertierung ist ein Zeiger auf die erste Zelle des Arrays. Da die erste Zelle dieselbe Adresse hat wie das Array, ist diese Konvertierung für Compiler trivial – es muss lediglich der Typ angepasst werden (siehe Abbildung 5.5).

Zeigerarithmetik

Es ist in begrenztem Rahmen möglich, mit Zeigern zu rechnen:

```
p + n   // Liefert einen Zeiger, der n Zellen hinter *p zeigt
p - n   // Liefert einen Zeiger, der n Zellen vor *p zeigt
q - p   // Liefert die Anzahl Zellen zwischen *p und *q
```

Diese Operationen sind ausschließlich innerhalb eines Arrays wohldefiniert. Bei allen anderen Speicherobjekten führt Zeigerarithmetik zu undefiniertem Verhalten.

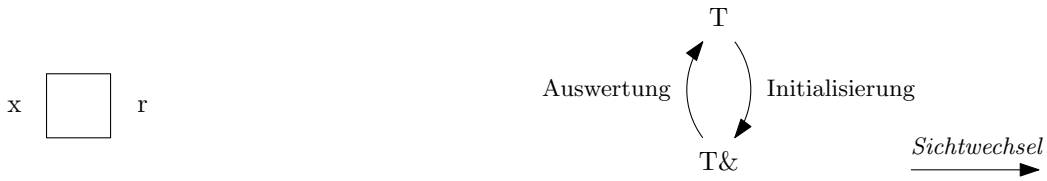

Abbildung 5.6: Referenzen als Alias für Speicherobjekte / Referenzen im Typsystem

5.3.3 Referenzen

Zeitgleich mit dem Überladen von Operatoren wurden Referenzen eingeführt, um die Parameterübergabe per Referenz zu unterstützen [Str94, 3.7]. Die Deklaration `T& r = x;` führt eine Referenz `r` auf ein bestehendes Speicherobjekt `x` ein. Konzeptionell sind `r` und `x` Aliase für dasselbe Speicherobjekt (siehe Abbildung 5.6 links).

Bei der Auswertung eines Ausdrucks vom Typ `T&` wird implizit eine Konvertierung nach `T` vorgenommen, deren Ergebnis das referenzierte Speicherobjekt ist [ISO03, 5 expr §6]. Die „Referenz als solche“ ist daher in Ausdrücken niemals zugreifbar. Folglich kann es auch keine Operationen auf Referenzen geben („Referenz-Arithmetik“ o.ä.).

Referenzen sind keine Zeiger

Tabelle 5.2 unterscheidet Zeiger von Referenzen. Insbesondere ist die Initialisierung bei Referenzen obligatorisch, und eine spätere Neubindung ist nicht vorgesehen. Weiterhin gibt es keine `null`-Referenz. Diese Eigenschaften sind für die Parameterübergabe per Referenz ideal. Für Verweissemantik sind diese Eigenschaften dagegen oft zu restriktiv.

Technisch werden Referenzen für gewöhnlich durch konstante, implizit dereferenzierte Zeiger realisiert. Oft können sie aber auch komplett wegoptimiert werden, zum Beispiel bei lokalen Referenzen oder inline-Prozeduren. Für den Programmierer sind diese Details jedoch irrelevant, Referenzen sind keine Zeiger. Insbesondere können Referenztypen nicht wie Zeigertypen geschachtelt werden, d.h. es gibt keine Referenzen auf Referenzen.

	Zeigervariable	**Benannte Referenz**
Ungebunden	T* p = 0;	✗
Gebunden	T* p = &x;	T& r = x;
Gebundenes Speicherobjekt	*p	r
Neubindung	p = &y;	✗
Speicherverbrauch	sizeof(T*)	unspezifiziert [ISO03, 8.3.2 dcl.ref §3]
Array von	T* ap[n];	✗
Zeiger auf	T** pp;	✗
Referenz auf	T*& rp;	✗
Dynamische Erzeugung	new T*	✗

Tabelle 5.2: Zeiger und Referenzen

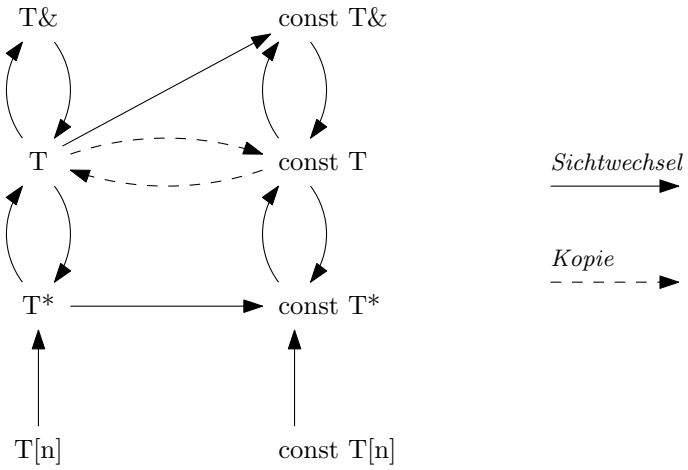

Abbildung 5.7: const correctness

5.3.4 const correctness

Mit dem reservierten Wort `const` kann das Typsystem die unbeabsichtigte Änderung von Speicherobjekten verhindern. Besonders interessant ist dabei der Zusammenhang zwischen Speicherobjekten und Zeigern bzw. Referenzen (siehe Abbildung 5.7).

Der Adressoperator führt von `T` zu `T*` und von `const T` zu const `T*`. Weiterhin existiert eine Konvertierung von `T*` zu `const T*`. Auf diese Weise können konstante Sichten auf veränderliche Speicherobjekte geschaffen werden. Veränderliche Sichten auf konstante Speicherobjekte sind dagegen nicht vorgesehen. Ähnliche Regeln gelten für Referenzen. Insbesondere binden Referenzen `T&` nicht an konstante Speicherobjekte `const T`.

Achtung: `const T*` bzw. `const T&` heißt lediglich, dass keine Veränderungen über den Zeiger bzw. die Referenz möglich sind. Ob das gebundene Speicherobjekt selbst konstant oder veränderlich ist, ist dem Zeiger bzw. der Referenz nicht anzusehen.

Konstante Operationen

Auf konstanten Exemplaren und über konstante Sichten auf Exemplare können ausschließlich *konstante Operationen* aufgerufen werden, welche mit dem reservierten Wort `const` als solche markiert sind. Aufrufe von verändernden Operationen auf konstanten Exemplaren sind nicht übersetzbar. Konstante Methoden können wiederum nicht das aktuelle Exemplar verändern. Um dies sicherzustellen, wird der Typ des `this`-Zeigers angepasst. In verändernden Methoden ist `this` vom Typ `T*`, in konstanten Methoden dagegen vom Typ `const T*` [ISO03, 9.3.2 class.this §1].

Verwendet ein Exemplar native Zeigervariablen als Felder, dann sind in konstanten Methoden lediglich die Zeigervariablen selbst konstant, nicht jedoch die referenzierten Speicherobjekte. Falls letztere zum logischen Zustand des Exemplars gehören, muss der Programmierer eigenständig darauf achten, diese nicht zu verändern.

Listing 5.1: Zeichenketten

```
1 const char* p   = "C++";        // p zeigt auf konstante Zeichenkette
2       char  a[] = "C++";        // a ist veränderliche Zeichenkette
3 std::cout << a << std::endl;    // C++
4
5 a[3] = ' ';                     // Terminator durch Leerzeichen ersetzt
6 std::cout << a << std::endl;    // C++ nx;)wb|V3F}cYcN!#AQ6+u5\BY/FBoA?AK!,~\Q2
7
8 a[1] = 0;                       // Terminator mitten im Array
9 std::cout << a << std::endl;    // C
```

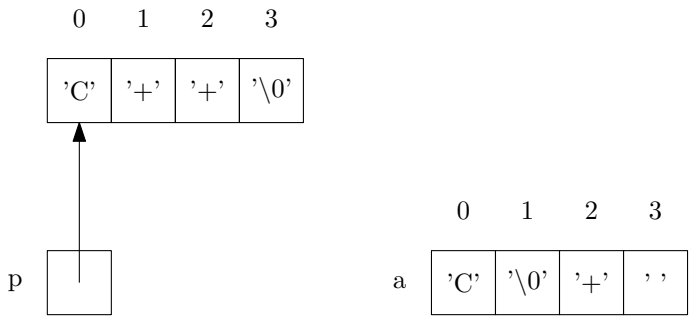

Abbildung 5.8: Zeichenketten

5.3.5 Zeichenketten

Die aus C übernommenen Zeichenketten (*C-Strings*) sind `char`-Arrays, welche mit dem unsichtbaren Zeichen `'\0'` abgeschlossen werden. Ohne diese terminierende `'\0'` wird in der Praxis über die Grenzen des Arrays hinweg auf den Speicher zugegriffen, bis zufällig eine `'\0'` im Speicher gefunden wird (siehe Zeilen 5 und 6 in Listing 5.1).

Stringliterale der Länge `n` sind vom Typ `const char[n+1]` (`n` sichtbare Zeichen plus die terminierende `'\0'`) und werden in statischen Speicherbereichen abgelegt. Daher können Zeiger auf Stringliterale problemlos herumgereicht werden. Stringliterale können auch als Initialisierer für `char`-Arrays verwendet werden. Die Zeichenkette wird dabei vollständig kopiert, und das deklarierte `char`-Array darf anschließend nach Belieben verändert werden. Schreibender Zugriff auf Stringliterale ist dagegen nicht erlaubt.

Bei der Manipulation von veränderlichen C-Strings obliegt die Speicherverwaltung den Klienten. Das führt in der Praxis oft zu Speicherverschwendung oder Pufferüberläufen. Eine komfortable und sichere Alternative ist `std::string`. Dieser Typ entlastet Klienten von der fehleranfälligen Speicherverwaltung und bietet viele sinnvolle Operationen an.

Der Konstruktor `std::string(const char*)` ermöglicht die Angabe von Stringliteralen an allen Stellen, an denen ein `std::string` erwartet wird. Die Konvertierung erfolgt in zwei Schritten: `const char[n+1]` → `const char*` → `std::string`.

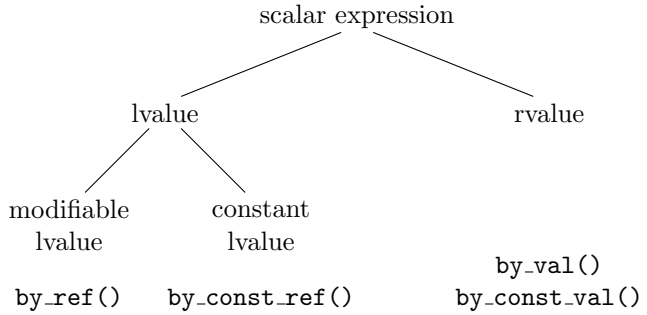

Abbildung 5.9: skalare Ausdrücke

5.3.6 lvalues und rvalues

C++ teilt die Menge der Ausdrücke zur Übersetzungszeit in *lvalues* und *rvalues* ein, d.h. jeder Ausdruck ist entweder ein lvalue oder ein rvalue [ISO03, 3.10 basic.lval §1]. Wenn man sagt „`procedure()` ist ein rvalue“, dann meint man damit nicht das Ergebnis des Prozeduraufrufs, welches im Allgemeinen erst zur Laufzeit feststeht, sondern den Ausdruck `procedure()` als syntaktisches Element im Quelltext. Zur Laufzeit gibt es weder lvalues noch rvalues. Trotzdem sagt man manchmal „`procedure` liefert einen rvalue“.

Skalare Typen

Bei skalaren Typen ist die Unterscheidung zwischen lvalues und rvalues trivial. Skalare lvalues sind Ausdrücke, deren Ergebnis ein skalares Speicherobjekt ist, zum Beispiel `i`, `array[0]` oder `*pointer`. Allgemein sind genau solche Ausdrücke lvalues, auf die der Adressoperator angewendet werden kann. Skalare rvalues liefern dagegen Werte oder Zeiger, zum Beispiel `42`, `3*i` oder `&(any_lvalue)`. Allgemein sind genau solche Ausdrücke rvalues, auf die der Adressoperator nicht angewendet werden kann.

Skalare lvalues, die eine Variable liefern, werden als *modifiable lvalues* bezeichnet. Dies sind die einzigen skalaren Ausdrücke, die auf der linken Seite der Zuweisung stehen dürfen. *Constant lvalues* liefern dagegen Konstanten. Bei skalaren rvalues gibt es diese Unterscheidung nicht, weil Werte und Zeiger sowieso unveränderlich sind. Eine zusätzliche Markierung mit `const` wäre sinnlos. Daher werden skalare rvalues als *modifiable rvalues* angesehen. Die folgenden Signaturen verdeutlichen alle skalaren Ausdruckskategorien:

```
      int  by_val();          // #1 Aufruf ist ein modifiable rvalue [3.10 §5]
const int  by_const_val();    // #2 Aufruf ist ein MODIFIABLE rvalue [3.10 §9]
      int& by_ref();          // #3 Aufruf ist ein modifiable lvalue [3.10 §3]
const int& by_const_ref();    // #4 Aufruf ist ein constant   lvalue [3.10 §3]
```

Das `const` im Rückgabetyp der zweiten Signatur wird ignoriert, weil es keine skalaren konstanten rvalues gibt. Die dritte Signatur ist beim Verändern von Sammlungen sinnvoll, zum Beispiel `numbers.at(i) = 42`. Die vierte Signatur wird selten manuell geschrieben, da die Rückgabe per konstanter Referenz bei skalaren Typen keinen Vorteil gegenüber der Rückgabe per Konstruktion hat. Sie wird aber häufig von Template-Code generiert.

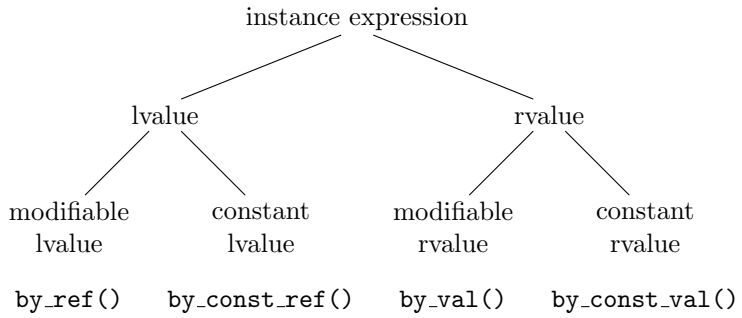

Abbildung 5.10: Exemplar-Ausdrücke

Exemplare

Die Unterscheidung zwischen lvalues und rvalues lässt sich bei Exemplaren am besten anhand eines kleinen Beispielprogramms erläutern:

```
std::string s("hello ");                                  // Zeile 1
for (int i = 0; i < 10; ++i)
{
    std::cout << s;                                       // Zeile 4
    std::cout << std::string("world") << std::endl;       // Zeile 5
}
```

In Zeile 1 wird ein `string`-Exemplar namens `s` erzeugt. In Zeile 4 wird insgesamt zehn Mal der Ausdruck `s` ausgewertet. Das Ergebnis dieser Auswertung ist dabei immer das in Zeile 1 erzeugte Exemplar. Deshalb ist der Ausdruck `s` ein lvalue. Allgemein sind solche Ausdrücke lvalues, die bei ihrer Auswertung ein bereits existentes Exemplar liefern.

Der `string`-Ausdruck in Zeile 5 liefert dagegen kein bereits existentes Exemplar. Jede Auswertung von `std::string("world")` bewirkt stattdessen die Erzeugung eines neuen, *temporären Exemplars*, welches am Ende von Zeile 5 automatisch zerstört wird. Deshalb ist `std::string("world")` ein rvalue. [1] Allgemein sind solche Ausdrücke rvalues, deren Auswertung die Erzeugung eines neuen, temporären Exemplars bewirkt.

Im Gegensatz zu den skalaren Typen unterscheidet man bei Exemplaren zwei Arten von rvalues. Eine Markierung des Rückgabetyps mit `const` ist hier semantisch relevant:

```
      T  by_val();          // #1 Aufruf ist ein modifiable rvalue [3.10 §5]
const T  by_const_val();    // #2 Aufruf ist ein CONSTANT   rvalue [3.10 §9]
      T& by_ref();          // #3 Aufruf ist ein modifiable lvalue [3.10 §3]
const T& by_const_ref();    // #4 Aufruf ist ein constant   lvalue [3.10 §3]
```

Über *modifiable rvalues* können sämtliche Operationen eines Exemplars aufgerufen werden, über *constant rvalues* dagegen nur konstante Operationen.

[1] Man beachte, dass im Ausdruck `std::string("world")` kein `new` vorkommt. Das ist ein Stolperstein für Programmierer aus anderen Sprachen. Der Ausdruck `new std::string("world")` wäre nicht vom Typ `std::string`, sondern vom Typ `std::string*`. Hierbei würde kein temporäres, sondern ein dynamisches Exemplar erzeugt werden, und das Ergebnis wäre ein Zeiger auf dieses dynamische Exemplar. Zeiger gehören aber zu den skalaren Typen und sind für diesen Abschnitt folglich irrelevant.

5.4 Parameterübergabe

Jeder formale Parameter wird vor Betreten des Prozedurrumpfs mit seinem entsprechenden aktuellen Parameter initialisiert, und zwar so, als ob man `T formal = actual;` geschrieben hätte. Die genaue Semantik der Parameterübergabe hängt vom Typ `T` ab.

Tabelle 5.3 stellt für vier gängige formale Parametertypen dar, welche Formen der Polymorphie bei der Bindung vorgesehen sind, und welche Varianten von Ausdrücken als aktuelle Parameter akzeptiert werden. `const T` und `const T&&` sind praktisch irrelevant.

Formaler	**Polymorphie**		**lvalue**		**rvalue**	
Parametertyp	**Coercion**	**Subtyping**	**modif.**	**const.**	**modif.**	**const.**
T	✓		✓	✓	✓	✓
T&		✓	✓			
const T&	✓	✓	✓	✓	✓	✓
T&& (C++0x)		✓			✓	

Tabelle 5.3: Parameterübergabemechanismen (modif = modifiable, const = constant)

5.4.1 Übergabe per Konstruktion (T)

Bei der Übergabe per Konstruktion ist ein formaler Parameter ein eigenständiges Speicherobjekt. Der aktuelle Parameter bestimmt den verwendeten Konstruktor und damit die Semantik der Parameterübergabe. Sind formaler und aktueller Parameter vom selben Typ, dann spreche ich von einer *Übergabe per Kopie* – es sei denn, der aktuelle Parameter ist ein rvalue, dann handelt es sich um eine *Übergabe per Transfer*. Weichen formaler und aktueller Parametertyp voneinander ab, dann findet eine *Übergabe per Coercion* statt:

```
void broadcast(std::string message);   //        T -> Übergabe per Konstruktion

std::string variable = "test";
broadcast(variable);                   // lvalue T -> Übergabe per Kopie

broadcast(std::string("test"));        // rvalue T -> Übergabe per Transfer

broadcast("test");                     //   kein T -> Übergabe per Coercion
```

5.4.2 Übergabe per Referenz (T&)

Bei der Übergabe per Referenz ist ein formaler Parameter ein Alias für den aktuellen Parameter. Veränderungen innerhalb der Prozedur wirken sich auf Klienten aus:

```
template <typename T> void swap(T& a, T& b)  // T& -> Übergabe per Referenz
{
    T c(a);
    a = b;   // verändert ersten aktuellen Parameter
    b = c;   // verändert zweiten aktuellen Parameter
}
```

Bei der Übergabe per Referenz werden ausschließlich modifizierbare lvalues des Typs `T` (oder eines Subtyps von `T`) akzeptiert und keine Coercions vorgenommen, weil ein Klient die Veränderungen ansonsten nicht beobachten könnte. Falls man keine Veränderungen beabsichtigt sondern lediglich die Kosten einer Kopie scheut, sollte man stattdessen die Übergabe per konstanter Referenz wählen (siehe nächster Abschnitt).

5.4.3 Übergabe per konstanter Referenz (const T&)

Die Übergabe per konstanter Referenz unterstützt neben Subtyping auch Coercion und akzeptiert sämtliche Varianten von Ausdrücken als aktuelle Parameter. Sie erlaubt dafür jedoch keine Veränderung der Parameter, da dies dem Gedanken von `const` widersprechen würde und im Zusammenhang mit rvalues und Coercion sinnlos und verwirrend wäre.

Skalare Werte haben keine Adressen und müssen daher temporär im Speicher abgelegt werden, bevor sie per konstanter Referenz übergeben werden können. Coercion wird über temporäre Speicherobjekte des jeweiligen Zieltyps realisiert. Beide Aspekte können anhand einer Prozedur mit drei konstanten Referenzparametern verdeutlicht werden:

```
void function(const int& tag, const std::string& monat, const int& jahr);

short s = 3;
function(s, "August", 1492);
```

Die ersten beiden aktuellen Parameter `s` und `"August"` sind vom falschen Typ (`short` statt `int` und `const char[7]` statt `std::string`), und der dritte Parameter `1492` ist ein skalarer Wert. Deswegen werden intern zunächst drei temporäre Speicherobjekte erzeugt, welche anschließend per konstanter Referenz übergeben werden:

```
int         temp1 = s;
std::string temp2 = "August";
int         temp3 = 1492;

function(temp1, temp2, temp3);
```

In der Praxis werden skalare Typen selten per konstanter Referenz übergeben. Das Kopieren skalarer Typen ist nicht teuer genug, um eine zusätzliche Indirektionsstufe zu rechtfertigen. Außerdem spielt Subtyping bei skalaren Typen keine Rolle.

Die Formulierung „per konstanter Referenz“ ist streng genommen falsch, da Referenzen grundsätzlich konstant sind (siehe Abschnitt 5.3.3). Eigentlich meint man „per Referenz auf konstantes Speicherobjekt“, wenn man „per konstanter Referenz“ sagt.

5.4.4 Übergabe per rvalue Referenz (T&&)

Der Parametertyp `T&&` bindet ausschließlich an modifizierbare rvalues. Die aufgerufene Prozedur kann das übergebene Speicherobjekt daher nach Belieben verändern, ohne dass ein Klient davon etwas mitbekommt. Dies ermöglicht eine sehr effiziente Realisierung von konzeptionell seiteneffektfreien Operationen (Details folgen in Abschnitt 6.3.3).

Der Typ `T&&` mag auf den ersten Blick wie eine Referenz auf eine Referenz aussehen. In Abschnitt 5.3.3 wurde aber bereits erläutert, dass solche Typen nicht existieren.

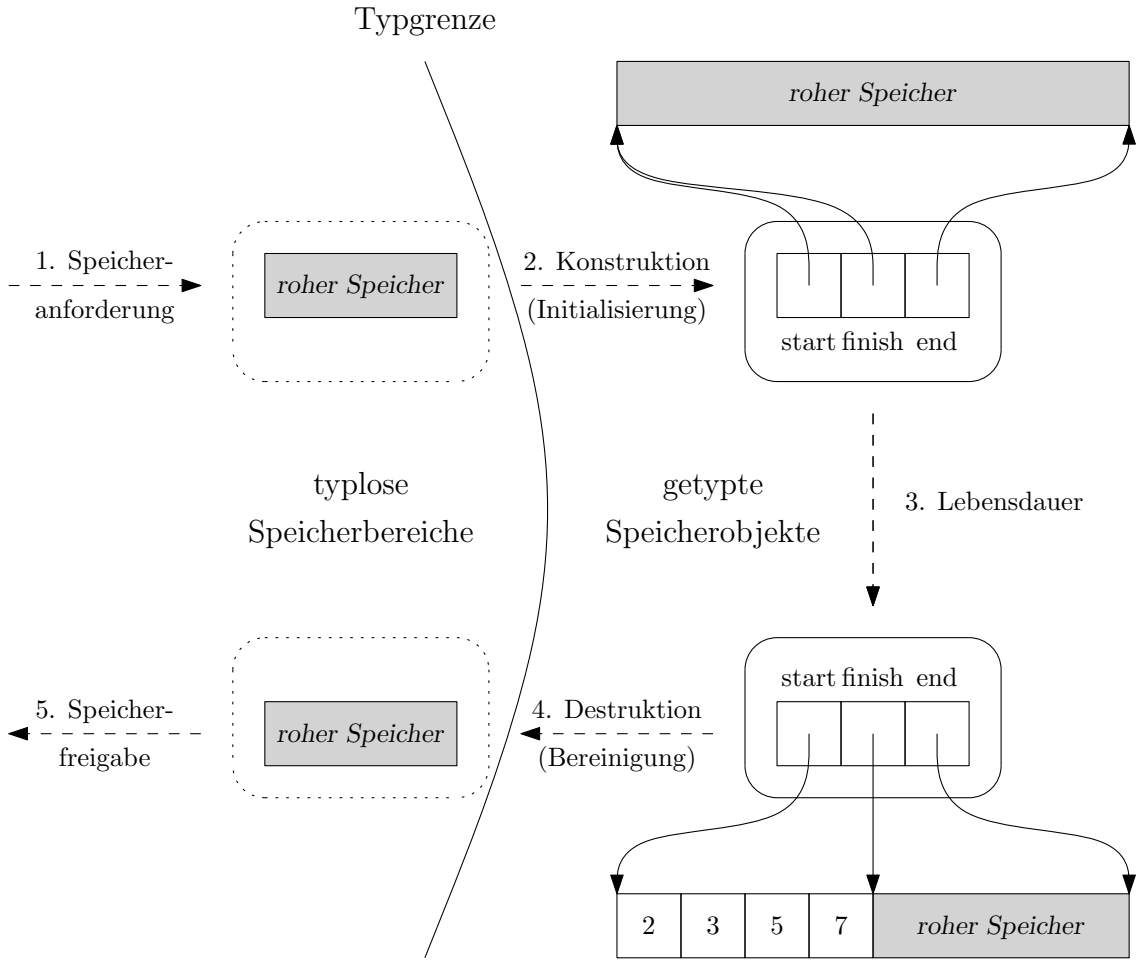

Abbildung 5.11: Der Lebenszyklus eines Speicherobjekts vom Typ `std::vector<int>`

5.5 Der Lebenszyklus von Speicherobjekten

Der Lebenszyklus von Speicherobjekten lässt sich in fünf Phasen einteilen: Speicheranforderung, Konstruktion, Lebensdauer, Destruktion, Speicherfreigabe. Abbildung 5.11 veranschaulicht diese Phasen am Beispiel `std::vector<int>`. Bei kompletten Speicherobjekten beginnt und endet der Lebenszyklus mit der Speicherverwaltung, auf die in dieser Arbeit nicht näher eingegangen wird. Für Komponenten (Zellen und Felder) entfällt die Speicherverwaltung, da jene direkt in ihr komplettes Speicherobjekt eingebettet sind.

Konstruktoren wandeln typlose Speicherbereiche in getypte Speicherobjekte um. Im Zuge dieser Initialisierung können weitere Ressourcen angefordert werden (zum Beispiel ein externer Speicherbereich wie in Abbildung 5.11 dargestellt). Die Lebensdauer beginnt mit Abschluss des Konstruktors und endet mit Beginn des Destruktors. Letzterer hat die Aufgabe, die im Konstruktor angeforderten Ressourcen wieder freizugeben.

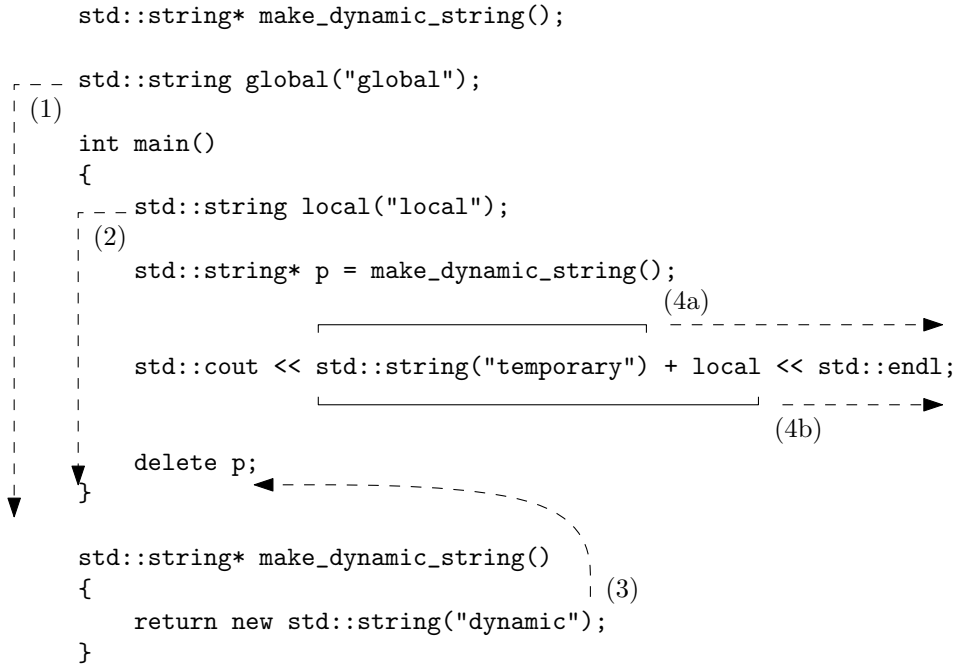

Abbildung 5.12: Lebensdauer kompletter Speicherobjekte

5.5.1 Ein genauerer Blick auf die Lebensdauer

Komplette Speicherobjekte

Abbildung 5.12 veranschaulicht die Lebensdauer kompletter Speicherobjekte. Die Pfeile beginnen mit Abschluss der Konstruktion und enden unmittelbar vor der Destruktion.

Globale Variablen (1) und statische Felder (Klassenvariablen) werden vor Eintritt in `main` konstruiert und nach Austritt aus `main` bereinigt. Lokale Variablen (2) leben normalerweise vom Durchlauf ihrer Deklaration bis zum Verlassen des Blocks. Falls sie mit `static` markiert sind, leben sie vom *ersten* Durchlauf ihrer Deklaration bis zum Ende des Programms. Die Lebensdauer dynamischer Speicherobjekte (3) wird zur Laufzeit durch die Operatoren `new` und `delete` bestimmt.

Temporäre Exemplare (4) leben von der Auswertung eines rvalues bis zum Ende des vollständigen Ausdrucks, in den dieser rvalue eingebettet ist. Bei der Ausgabe in `main` treten zwei temporäre Exemplare auf. Das erste (4a) wird durch Auswertung des Ausdrucks `std::string("temporary")` erzeugt, das zweite (4b) durch die Konkatenation. Am Ende der Zeile werden beide temporären Exemplare wieder zerstört.

Komponenten

Felder werden vor Eintritt in den Konstruktorrumpf initialisiert und nach Austritt aus dem Destruktorrumpf in umgekehrter Deklarationsreihenfolge bereinigt. Zellen werden in aufsteigender Reihenfolge initialisiert und in absteigender Reihenfolge bereinigt.

5.6 Zusammenfassung

Dieses Kapitel warf einen detaillierten Blick auf einige fundamentale Sprachmechanismen von C++, die im Zusammenhang mit Werttypen relevant sind. Insbesondere werden *const correctness* und die Unterscheidung von Ausdrücken in *lvalues* und *rvalues* im nächsten Kapitel eine zentrale Rolle spielen.

Die Lebenszyklen von Speicherobjekten sind in C++ deutlich komplexer als in anderen Sprachen. Als C++ Programmierer muss man sich ständig Gedanken darüber machen, wer ein Speicherobjekt „besitzt“ und damit für dessen Zerstörung verantwortlich ist. Eine dynamische Erzeugung mittels `new` und ein indirekter Zugriff über Zeiger sind in C++ daher eher die Ausnahme als die Regel. Insbesondere ist der gemeinsame Zugriff auf dynamisch erzeugte, unveränderliche Exemplare in C++ nicht idiomatisch.

6 Werttypen in C++

Nachdem die fundamentalen Sprachmechanismen von C++ im letzten Kapitel geklärt wurden, kann das Wertmodell nun auf C++ angewendet werden. Dabei untersuche ich, welche Werttypen bereits in C++ existieren, und wie benutzerdefinierte Werttypen in C++ umgesetzt werden können. Dabei wird unterschieden zwischen *flachen Werttypen*, deren Variablen die komplette Repräsentation ihres gebunden Werts enthalten, und *tiefen Werttypen*, bei denen Teile der Wertrepräsentation ausgelagert werden.

Anschließend fasse ich die gewonnenen Erkenntnisse zur Implementierung und Verwendung von Werttypen in Form eines „Kochrezepts“ zusammen. Zuletzt bewerte ich, wie gut C++ sich mit dem Wertmodell verträgt.

6.1 Vordefinierte Werttypen

In diesem Abschnitt untersuche ich kurz die primitiven Datentypen und diskutiere anschließend, warum Zeichenketten in C++ keine Werttypen sind und warum in C++ kein Bedarf für Wrapperklassen (wie sie in Java existieren) besteht.

6.1.1 Primitive Werttypen

Abbildung 6.1 zeigt einen vereinfachten Ausschnitt aus dem Typsystem von C++. Die arithmetischen Typen entsprechen weitestgehend den primitiven Typen anderer imperativer Programmiersprachen und können als Werttypen aufgefasst werden. Primitive Werte werden über getypte Literale aus einer festen Wertemenge ausgewählt, und die Operationen auf primitiven Werten sind seiteneffektfrei und referenziell transparent.

Variablen

Neben den Operationen auf primitiven Werten gibt es auch Operationen auf Variablen primitiver Werttypen, zum Beispiel `++` (Präfix und Postfix) oder `&`. Man kann mehrere Zeiger auf dieselbe `int`-Variable halten und Änderungen daran beobachten:

```
int* p = new int(42);
int* q = p;
++*p;
assert(*q == 43);
delete p;
```

Hierbei wird allerdings nicht der Wert 42 geändert, was prinzipiell unmöglich ist, sondern lediglich eine `int`-Variable an einen anderen Wert gebunden. Der Programmierer ist selbst dafür verantwortlich, die Verwendung von Aliasen sinnvoll einzuschränken.

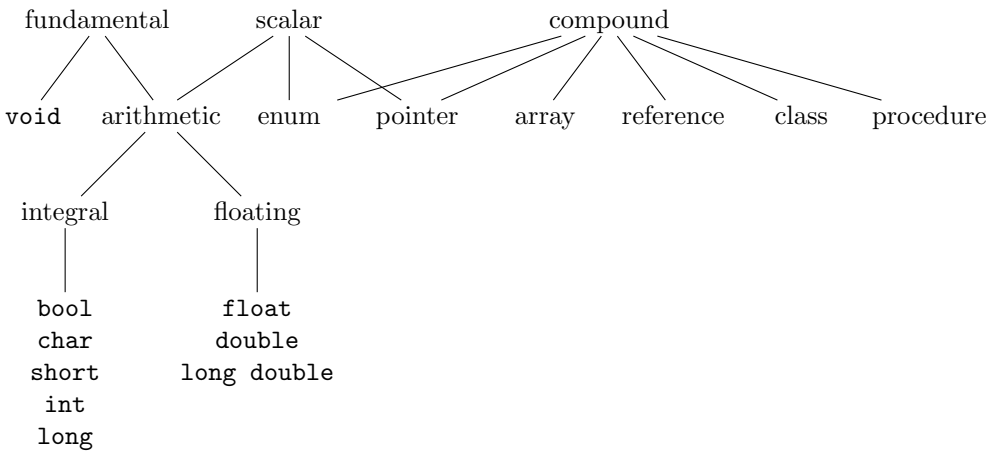

Abbildung 6.1: Vereinfachter Ausschnitt aus dem Typsystem von C++

6.1.2 Zeichenketten

Im Gegensatz zu Java oder C# sind Zeichenketten in C++ Objekttypen, d.h. `std::string` verfügt über modifizierende Operationen:

```
std::string a = "hello world";
std::string b = a;              // b ist unabhängige Kopie von a
a.append("!");
assert(a == "hello world!");    // a wurde verändert
assert(b == "hello world");     // b blieb unverändert
```

Die Tatsache, dass `append` einen `string` verändert anstatt einen anderen `string` zurückzugeben, widerspricht dem Wertmodell. Aufgrund der Kopiersemantik von `std::string` wirken sich Änderungen auf `a` allerdings nicht auf `b` aus und umgekehrt. Es gibt keinen gemeinsamen Zustand, der von mehreren `string`-Variablen geteilt wird. Genau wie im Wertmodell kann man Änderungen also nur beobachten, wenn man Zugriff auf die veränderte Variable hat.

Für Typen mit Kopiersemantik gibt es in C++ zwei Bezeichnungen: „value types“ und „concrete types“ [Str00, 10.3.4]. Da der Begriff „Werttyp“ in dieser Arbeit bereits belegt ist, werde ich Typen mit Kopiersemantik im folgenden als *Konkrete Typen* bezeichnen.

6.1.3 Wrapperklassen

In Programmiersprachen, die ausschließlich Referenztypen als generische Typparameter zulassen, benötigt man für primitive Werttypen sog. *Wrapperklassen*. In C++ gibt es diese Beschränkung nicht, wodurch der Bedarf nach solchen Wrapperklassen entfällt, d.h. Sammlungen können problemlos mit primitiven Werttypen parametrisiert werden:

```
std::list<int> numbers = {4, 8, 15, 16, 23};
numbers.push_back(42);
numbers.remove_if(is_odd);
```

6.2 Aufzählungswerttypen

Der Typkonstruktor `enum` wurde leicht modifiziert aus C übernommen. Die Elemente werden durch Aufzählung benannt und sind anschließend über diese Namen auswählbar:

```
enum farbe
{
    KARO, HERZ, PIK, KREUZ
};
```

Intern werden die Elemente auf Ganzzahlen abgebildet, zu denen implizit gewandelt werden kann. Daher funktionieren alle bekannten Vergleichsoperatoren auch auf `enum`s. Außerdem ist diese implizite Wandlung beim Zugriff auf Arrays nützlich:

```
int wertigkeit[] = {9, 10, 11, 12};
farbe trumpf = HERZ;
int punkte = wertigkeit[trumpf];   // HERZ -> 1
```

Die Operationen eines `enum` werden als freistehende Funktionen realisiert:

```
farbe naechste_farbe(farbe x)
{
    return farbe((x + 1) & 3);
}
```

Die Unveränderlichkeit der Aufzählungselemente wird vom Compiler garantiert, d.h. `KARO`, `HERZ`, `PIK` und `KREUZ` können nicht verändert werden. Seiteneffektfreiheit und referenzielle Transparenz müssen dagegen vom Programmierer sichergestellt werden.

6.2.1 Namensräume

Die Elemente eines Aufzählungstyps haben keinen eigenen Namensraum, sondern gehören dem umgebenden Namensraum an. Gleichnamige Elemente verschiedener Aufzählungstypen im selben Namensraum kollidieren also:

```
enum organ
{
    LUNGE, HERZ, LEBER, NIERE   // error: conflicting declaration of 'HERZ'
};
```

Um dieses Problem zu umgehen, können Aufzählungstypen in eigenen Namensräumen geschachtelt werden:

```
namespace organ
{
    enum organ
    {
        LUNGE, HERZ, LEBER, NIERE
    };
}

organ::organ pumpe = organ::HERZ;
```

In C++0x gibt es einen neuen Typkonstruktor namens `enum class`, der das Namensraumproblem löst und außerdem keine impliziten Wandlungen mehr zulässt:

```
enum class farbe
{
    KARO, HERZ, PIK, KREUZ
};

farbe trumpf = farbe::HERZ;

enum class organ
{
    LUNGE, HERZ, LEBER, NIERE
};

organ pumpe = organ::HERZ;
```

Im Gegensatz zu einer gewöhnlichen Klasse können im Rumpf einer Aufzählungsklasse keine Operationen definiert werden. Diese sind weiterhin als freistehende Funktionen zu realisieren.

6.2.2 Variablen

Neben den Operationen auf den unveränderlichen Werten eines Aufzählungstyps können auch Operationen auf den Variablen eines Aufzählungstyps definiert werden, zum Beispiel Präfix- und Postfix-Inkrement. Dabei ist es üblich, die Postfix-Variante auf die Präfix-Variante zurückzuführen:

```
// Präfix: Ändert Variable und liefert geänderte Variable
farbe& operator++(farbe& v)
{
    v = naechste_farbe(v);
    return v;
}

// Postfix: Ändert Variable und liefert alte Belegung
farbe operator++(farbe& v, int unused_postfix_marker)
{
    farbe kopie(v);
    ++v;
    return kopie;
}
```

Auch hier werden keine Werte verändert, sondern lediglich Variablen an andere Werte gebunden. Operationen auf Variablen erkennt man daran, dass sie den entsprechenden Parameter per (nicht-konstanter) Referenz entgegennehmen.

Hervorzuheben sind die unterschiedlichen Rückgabetypen. Die Präfix-Variante kann einfach die geänderte Variable liefern, während bei der Postfix-Variante keine Variable existiert, die den gewünschten Wert nach Abschluss der Prozedur enthält, denn `v` ist zu diesem Zeitpunkt schon verändert, und die lokale Variable `kopie` wurde bereits zerstört.

6.3 Zusammengesetzte flache Werttypen

C++ verfügt über zwei Typkonstruktoren für zusammengesetzte Typen, die sich nur unwesentlich in Sichtbarkeits- und Vererbungsdetails unterscheiden: `class` und `struct`. Üblicherweise verwendet man `struct` für einfache Datencontainer und `class` für gekapselte Exemplare. Dies ist jedoch lediglich eine Konvention.

6.3.1 Ein erster naiver Ansatz

C++ bietet keinen Typkonstruktor für zusammengesetzte Werttypen. Stattdessen müssen sie durch Objekttypen simuliert werden. Diese Simulation weicht jedoch aufgrund der zugrundeliegenden Direktsemantik von anderen Sprachen ab. Ein erster naiver Versuch einer Wertklasse `point` könnte wie folgt aussehen:

```
class point
{
    const int x_;
    const int y_;

public:

    point(int x, int y)       // Kopf
    : x_(x), y_(y)            // Initialisierungsliste
    {}                        // Rumpf

    int x()
    {
        return x_;
    }

    int y()
    {
        return y_;
    }
};

int main()
{
    point p(1, 2);            // Variable
    const point q(3, 4);      // Konstante
    p = q;                    // Problem 1: Zuweisung
    assert(p.x() == q.x());   // Problem 2: Konstanten
}
```

Die Felder eines Exemplars werden in der *Initialisierungsliste* initialisiert, welche durch einen Doppelpunkt vom Konstruktorkopf getrennt ist. Innerhalb des Konstruktorrumpfs können keine Initialisierungen durchgeführt werden. Die Zuweisung `x_ = x` wäre in diesem Beispiel nicht übersetzbar, weil Zuweisungen auf Konstanten verboten sind.

Der gezeigte naive Ansatz hat allerdings zwei Probleme:

Problem 1: Zuweisung

Die Zuweisung `p = q` in `main` scheitert wegen der konstanten Felder:

```
In member function 'point& operator=(const point&)':
error: const member 'const int x_', can't use default assignment operator
error: const member 'const int y_', can't use default assignment operator
note: synthesized method 'point& operator=(const point&)' first required here
```

Wie man anhand der Fehlermeldung erahnen kann, ist die Zuweisung `p = q` lediglich syntaktischer Zucker für den Aufruf der Operation `p.operator=(q)`, welche bei Bedarf automatisch vom Compiler generiert wird. Dabei werden einfach die Felder zugewiesen:

```
point& operator=(const point& that)
{
    x_ = that.x_;   // error: assignment of read-only data-member
    y_ = that.y_;   // error: assignment of read-only data-member
    return *this;
}
```

Damit die Zuweisung auf Variablen eines Werttyps funktioniert, dürfen die Felder einer Wertklasse folglich niemals mit `const` markiert werden.

Problem 2: Konstanten

Das zweite Problem ist der Aufruf `q.x()`, welcher ebenfalls nicht übersetzbar ist:

```
error: no matching function for call to 'point::x() const'
```

Da `q` eine Konstante ist, erlaubt der Compiler ausschließlich Aufrufe von konstanten Operationen (siehe Seite 26). Daher sollten alle Wertoperationen mit `const` markiert werden, da es ansonsten in vielen Kontexten verboten wäre, sie aufzurufen.

Lösung

Die Felder eines Werttyps dürfen nicht mit `const` markiert werden, damit Zuweisungen funktionieren, und Wertoperationen müssen mit `const` markiert werden, damit Wertoperationen auf Konstanten aufgerufen werden können. Eine überarbeitete Fassung des ersten naiven Ansatzes könnte demnach wie folgt aussehen:

```
class point
{
    int x_;
    int y_;

public:
    point(int x, int y) : x_(x), y_(y) {}

    int x() const { return x_; }

    int y() const { return y_; }
};
```

6.3.2 Flüchtige Speicherobjekte

Ein wesentlicher Aspekt wertorientierter Programmierung ist das Auswerten von Ausdrücken, wobei beliebig viele Eingaben schrittweise auf ein Ergebnis reduziert werden. Beispielsweise wird bei der Auswertung von `a + b + c` zunächst das Zwischenergebnis des Teilausdrucks `a + b` bestimmt. Konzeptionell handelt es sich dabei um einen Wert. Technisch muss eine Repräsentation dieses Wertes in einem flüchtigen Speicherobjekt abgelegt werden, damit die zweite Addition Zugriff darauf hat. Die Notwendigkeit dieser flüchtigen Speicherobjekte wurde auch schon von MacLennan beschrieben:

> Clearly, whenever a value is to be manipulated in a computer, it must be represented as the state of some physical [storage] object. Typically, there will be many such representing [storage] objects in a computer at a time. For instance, 2 can be represented by a bit pattern in a register and in several memory locations. Therefore, everything in a computer is [a storage] object; there are no values in computers. [Mac82, 4]

Bei der Simulation von zusammengesetzten Werttypen in C++ spielen temporäre Exemplare (Seite 33) die Rolle flüchtiger Speicherobjekte. Sie werden beim Auswerten von rvalues erzeugt und sind im lexikalischen Kontext ihrer Erzeugung anonym und identitätlos, weil der Adressoperator ausschließlich auf lvalues angewendet werden darf [ISO03, 5.3.1 expr.unary.op §2].

C++0x bietet bei der Parameterübergabe erstmals die Möglichkeit, zwischen lvalues und rvalues zu unterscheiden. Dadurch können temporäre Exemplare von Klienten nach Belieben verändert werden, ohne dass sich aus Klientensicht Seiteneffekte ergeben. Den zugrundliegenden Sprachmechanismus werde ich an Beispielen verdeutlichen, weil dieser beim Verständnis von rvalues, temporären Exemplaren und Werten hilft.

6.3.3 Weitere Wertoperationen

Bisher wurden nur Wertoperationen betrachtet, die `int` zurückliefern. Im folgenden werden Wertoperationen implementiert, in deren Signatur der Typ `point` auftaucht.

Vergleichsoperationen

Die Vergleichsoperatoren `==` und `!=` haben für benutzerdefinierte Typen keine vordefinierte Bedeutung. Insbesondere werden dabei keine Adressen verglichen. Sie müssen manuell überladen werden, wobei einer der beiden auf den anderen zurückführbar ist:

```
bool operator==(const point& a, const point& b)
{
    return (a.x() == b.x()) && (a.y() == b.y());
}

bool operator!=(const point& a, const point& b)
{
    return !(a == b);
}
```

Der formale Parametertyp `const point&` erlaubt die Verwendung sämtlicher `point`-Ausdrücke als aktuelle Parameter:

```
point p(1, 2);
const point q(3, 4);

assert(p != q);                          // 1
assert(p == point(1, 2));                // 2
assert(point(3, 4) == point(3, 4));      // 3
```

Alternativ wäre auch der formale Parametertyp `point` möglich gewesen. Hier muss man zwischen einer zusätzlichen Indirektionsstufe und einer unnötigen Kopie abwägen. Für Klienten ergibt sich kein semantischer Unterschied. Bei der Übergabe per Referenz wäre nur der erste Vergleich übersetzbar, bei der Übergabe per rvalue Referenz nur der dritte.

Die formalen Parametertypen werden noch einmal in Tabelle 6.1 zusammengefasst.

Formaler	**Polymorphie**		**lvalue**		**rvalue**	
Parametertyp	**Coercion**	**Subtyping**	**modif.**	**const.**	**modif.**	**const.**
T	✓		✓	✓	✓	✓
T&		✓	✓			
const T&	✓	✓	✓	✓	✓	✓
T&& (C++0x)		✓			✓	

Tabelle 6.1: Parameterübergabemechanismen (modif = modifiable, const = constant)

Addition von Punkten

Die Addition von Punkten kann durch Überladen des Operators `+` realisiert werden:

```
point operator+(const point& a, const point& b)
{
    return point(a.x() + b.x(), a.y() + b.y());
}
```

Durch die Auswertung des rvalues `point(p.x() + q.x(), p.y() + q.y())` wird ein temporäres `point`-Exemplar erzeugt, in welchem das Ergebnis der Addition abgelegt wird. Dieses temporäre Exemplar wird im Zuge der Rückgabe kopiert und anschließend zerstört. Seine Identität spielt für Klienten keine Rolle. Die Identität des temporären Exemplars auf Klientenseite ist wiederum für `operator+` nicht einsehbar. Konzeptionell wird also kein Exemplar zurückgegeben, sondern ein Wert.

Primitive Werttypen bieten die Kurzschreibweise `a += b`, um einen Wert direkt auf eine Variable addieren zu können. Um dies auch für `point`-Variablen zu ermöglichen, könnte dieser Operator basierend auf `+` und `=` implementiert werden:

```
point& operator+=(point& a, const point& b)
{
    a = a + b;
    return a;
}
```

Der erste Parametertyp `point&` akzeptiert nur lvalues als aktuelle Parameter. Es ist zum Beispiel nicht möglich, `point(1, 2) += p` zu schreiben. Man mache sich klar, dass der Beispiel-Ausdruck `p += q` und der darin enthaltene Teilausdruck `p` sowie der Ausdruck `a` drei verschiedene lvalues sind, die dasselbe `point`-Exemplar bezeichnen.

Der Rückgabetyp ist nicht `point` sondern `point&`, weil das Ergebnis von `p += q` nicht die Summe von `p` und `q` ist, sondern die Variable `p`.

Kombinierte Zuweisungen in Methoden implementieren

Es ist unnötig ineffizient, `operator+=` als freistehende Prozedur zu implementieren, weil dabei mehrere temporäre Exemplare erzeugt und zerstört werden. Daher bietet es sich an, kombinierte Zuweisungsoperatoren als verändernde Methoden zu implementieren:

```
// innerhalb der Klassendefinition von point
point& operator+=(const point& b)
{
    x_ += b.x_;
    y_ += b.y_;
    return *this;
}
```

Man beachte, dass diese Methode nicht mit `const` markiert ist. (Ansonsten wären die enthaltenen Zuweisungen nicht übersetzbar.) Sie ist also keine Wertoperation und kann nicht auf Konstanten aufgerufen werden. Aufgrund einer Sonderregel [ISO03, 3.10 §10] können verändernde Methoden auf rvalues aufgerufen werden, d.h. `point(1, 2) += p` ist jetzt übersetzbar. Widerspricht dies der Unveränderlichkeit von Werten? Nein. Hier wird ein temporäres Exemplar erzeugt und verändert. Ein Klient kann diese Änderung aber unter keinen Umständen bemerken. Aus seiner Sicht sind `point(1, 2) + p` und `point(1, 2) += p` semantisch äquivalent (aber die zweite Variante ist effizienter).

Um Code-Duplizierung zu vermeiden, kann `operator+` auf `operator+=` zurückgeführt werden:

```
point operator+(const point& a, const point& b)
{
    point copy(a);
    copy += b;
    return copy;
}
```

Aufgrund der erwähnten Sonderregel kann dies auch kompakter formuliert werden:

```
point operator+(const point& a, const point& b)
{
    return point(a) += b;
}
```

Dabei ruft `point(a)` den Kopierkonstruktor auf und liefert ein temporäres Exemplar, welches anschließend per `+=` verändert wird. Dieses temporäre Exemplar wird im Zuge der Ergebnisrückgabe kopiert, bevor es wieder zerstört wird.

Transparente Änderung temporärer Exemplare

In C++0x kann eine zweite Überladung des Operators + definiert werden:

```
point operator+(point&& a, const point& b)
{
    return a += b;
}
```

Diese Überladung wird im Zuge der *overload resolution* genau dann ausgewählt, wenn der linke Operand ein modifizierbarer rvalue ist, zum Beispiel bei `point(1, 2) + p`. Nun kann `a` direkt verändert werden, ohne dass ein Klient diese Änderung bemerken könnte. Bei einfachen Werttypen wie `point` lohnen sich solche Optimierungen noch nicht, aber für das Verständnis von rvalues ist diese Überlegung bereits recht hilfreich.

Theoretisch käme als Rückgabetyp auch `point&&` in Frage, wobei `return a += b;` durch `return static_cast<point&&>(a += b);` ersetzt werden müsste, weil `a += b` ein lvalue ist (und rvalue Referenzen nicht an lvalues binden). Dann würde ein Klient sein erstes übergebenes `point`-Exemplar als Ergebnis erhalten anstatt einer unnötigen Kopie. Aufrufe von Funktionen, die ihr Ergebnis per rvalue Referenz liefern, gehören allerdings einer dritten Ausdruckskategorie namens *xvalue* an. Um die vorliegende Diskussion nicht unnötig zu verkomplizieren, gehe ich in dieser Arbeit nicht näher auf xvalues ein.

Da die Addition von Punkten kommutativ ist, bietet es sich an, weitere Überladungen bereitzustellen, falls der zweite Operand ein rvalue ist (oder beide Operanden rvalues sind). Diese Möglichkeiten sollen hier jedoch nicht weiter vertieft werden.

6.3.4 Werte als Zeichenfolgen darstellen

Um Werte in eine menschenlesbare Form zu konvertieren, sehen viele Sprachen eine Operation vor, die einen String zurückliefert. Stattdessen gibt es in C++ die Konvention, den Stromausgabeoperator << zu überladen:

```
std::ostream& operator<<(std::ostream& os, const point& p)
{
    os << '(' << p.x() << ", " << p.y() << ')';
    return os;
}
```

Der formale Parameter `os` muss per Referenz übergeben werden, weil der Ausgabestrom manipuliert werden soll. Die Rückgabe per Referenz ermöglicht verkettete Ausgaben:

```
std::cout << point(1, 2) << " und " << point(3, 4) << " sind Punkte.\n";
```

Aufbauend auf `operator<<` sind auch seiteneffektfreie Konvertierungen möglich, indem man die Ausgabe auf einem `stringstream` vornimmt und daraus einen `string` extrahiert:

```
#include <sstream>

std::stringstream ss;
ss << point(1, 2);
std::string str = ss.str();
```

In der Boost-Bibliothek gibt es ein Template, das dieses Idiom kapselt:

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

std::string str = boost::lexical_cast<std::string>(point(1, 2));
```

6.3.5 Zusammenfassung

Eine in C++ simulierte Wertklasse beschreibt das Verhalten von Variablen, Konstanten und temporären Exemplaren, die an abstrakte Werte gebunden sind. Dies hat Auswirkungen auf das Design von Wertklassen. Aus der Veränderbarkeit von Variablen folgt, dass die Felder auf keinen Fall mit `const` markiert sein dürfen. Die Wertoperationen müssen wiederum mit `const` markiert werden, damit sie auf Konstanten aufrufbar sind.

Der Umgang mit Werten besteht primär in der Auswertung von rvalues. Dabei werden temporäre Exemplare erzeugt, in denen die Ergebnisse abgelegt werden. Klienten sind von der Erzeugung, Veränderung und Zerstörung dieser temporären Exemplare komplett abgeschirmt. Sie nehmen ausschließlich die darin repräsentierten Werte wahr. Die explizite Erzeugung von Exemplaren auf dem Heap ist bei Werttypen nicht sinnvoll.

Wertoperationen verwenden die Übergabe per Konstruktion oder konstanter Referenz. Aus Effizienzgründen kann man in C++0x zwei Überladungen pro Parameter anbieten, einmal per konstanter Referenz und einmal per rvalue Referenz. Die Rückgabe geschieht immer per Konstruktion.

Kombinierte Zuweisungsoperatoren erlauben die effiziente Veränderung von Variablen. Um Werte für Menschen lesbar zu machen, sollte der Operator << überladen werden.

6.4 Zusammengesetzte tiefe Werttypen

Die `point`-Exemplare aus dem letzten Abschnitt enthalten die komplette Repräsentation ihres gebundenen Werts. Bei Werttypen mit potentiell unendlich großen Wertemengen (zum Beispiel Zeichenketten) müssen dagegen Teile der Wertrepräsentation ausgelagert werden. Zur Veranschaulichung wird in diesem Abschnitt eine eigene `string`-Klasse mit manueller Speicherverwaltung entworfen. Dabei wird untersucht, was beim Erzeugen, Kopieren, Zuweisen, Vertauschen und Zerstören von `string`-Exemplaren geschieht. Später wird die Speicherverwaltung einem Standardcontainer übertragen und die `string`-Klasse um sinnvolle Operationen ergänzt: Indizierung einzelner Zeichen, Konkatenation und die Bildung von Teilstrings.

6.4.1 Manuelle Speicherverwaltung

Um Zeichenketten beliebiger Länge verarbeiten zu können, benötigt eine minimale `string`-Klasse einen Zeiger auf einen dynamisch angeforderten Zeichenpuffer. Die Länge wird in einem separaten Feld gespeichert:

```
class string
{
    char* buffer;
    int length;
```

(Die Klassendefinition wird in den folgenden Abschnitten ergänzt.)

Initialisierung

Um `string`-Exemplare mit C-Strings initialisieren zu können, ist ein passender Konstruktor erforderlich. So werden Deklarationen der Form `string a("Hund");` ermöglicht:

```
public:

    string(const char* p = "")
    {
        length = strlen(p);
        buffer = new char[length];
        memcpy(buffer, p, length);
    }
```

Der Konstruktor bestimmt zunächst die Länge des übergebenen C-Strings. Dann wird ein entsprechend großer Zeichenpuffer auf dem Heap reserviert, in welchen die Zeichen per `memcpy` hineinkopiert werden. Der 0-Terminator muss dabei nicht mitkopiert werden, weil die Längenangabe bereits im Feld `length` gespeichert ist.

Man beachte, dass eine Kopie des Zeigers `p` nicht ausreichend wäre. Eine solche flache Kopie würde nur bei der Übergabe von Literalen funktionieren. Der formale Parametertyp `const char*` bindet aber auch an `char*`, `char[n]` und `const char[n]`. Da die `string`-Klasse keinerlei Kontrolle über die Veränderung und Lebensdauer externer Arrays hat, muss die Zeichenkette im Konstruktor auf jeden Fall tief kopiert werden.

Falls ein Klient keinen aktuellen Parameter übergibt, wird der Default-Parameter `""` verwendet. Innerhalb des Konstruktors wird folglich ein dynamisches Array der Größe 0 angelegt. Alternativ hätte man auch einen parameterlosen Default-Konstruktor `string()` schreiben können, der kein Array anfordert und beide Felder mit `0` initialisiert.

Bereinigung

Der angeforderte Zeichenpuffer muss im Destruktor explizit zerstört werden, weil die Destruktoren von Zeigervariablen wirkungslos sind:

```
    ~string()
    {
        delete[] buffer;
    }
```

Der Destruktor `~string()` wird jedes Mal aufgerufen, wenn die Lebensdauer eines `string`-Exemplars endet. Aus diesem Grund dürfen sich mehrere `string`-Exemplare nicht denselben Zeichenpuffer teilen. Im Destruktor wäre schlicht nicht feststellbar, ob der Zeichenpuffer noch von anderen `string`-Exemplaren benötigt wird oder nicht.

Kopierkonstruktor

Bei der Initialisierung eines `string`-Exemplars mit einem anderen `string`-Exemplar wird der Kopierkonstruktor verwendet, zum Beispiel bei `string b(a);` oder `string b = a;`

```
string(const string& that)
{
    length = that.length;
    buffer = new char[length];
    memcpy(buffer, that.buffer, length);
}
```

Die Länge ist bereits bekannt und muss nur noch übernommen werden. Erneut findet eine tiefe Kopie statt. Dadurch wird sichergestellt, dass die beiden `string`-Exemplare unabhängig voneinander sind.

Kopierzuweisung

Der Zuweisungsoperator wird während der Lebensdauer eines Exemplars aufgerufen und ist schwieriger zu implementieren als der Kopierkonstruktor, weil der aktuelle Zustand des Ziel-Exemplars berücksichtigt werden muss:

```
string& operator=(const string& that)
{
    if (length != that.length)
    {
        char* p = new char[that.length];
        delete[] buffer;
        buffer = p;
        length = that.length;
    }
    memcpy(buffer, that.buffer, length);
    return *this;
}
```

Zunächst wird geprüft, ob die Länge des eigenen Zeichenpuffers exakt mit der Länge des zuzuweisenden Zeichenpuffers übereinstimmt. Dann können die bisher gespeicherten Zeichen einfach überschrieben werden. Ansonsten wird vorher ein neuer Zeichenpuffer passender Länge reserviert und der alte gelöscht. Falls die Reservierung wegen Speichermangel fehlschlägt, wird das `string`-Exemplar aufgrund der Reihenfolge der Anweisungen im selben Zustand verlassen wie zu Beginn der Zuweisung.

Der Zusammenhang zwischen Exceptions und Invarianten wurde erstmals gründlich in [Car94] erforscht und ist mittlerweile unter dem Begriff *exception safety* wohlbekannt. Eine knappe Erläuterung für Java-Programmierer befindet sich in [Blo08, Item 64].

Ausgabe

Der Ausgabeoperator << kann besonders effizient realisiert werden, wenn dieser Zugriff auf die Interna eines `string`-Exemplars hat. Zu diesem Zweck können Prozeduren innerhalb des Klassenrumpfs als `friend` deklariert werden:

```
    friend std::ostream& operator<<(std::ostream& os, const string& str);

};  // Ende der Klassendefinition von string

std::ostream& operator<<(std::ostream& os, const string& str)
{
    os.write(str.buffer, str.length);
    return os;
}
```

Man beachte, dass es unmöglich ist, `operator<<` als Methode in `string` zu realisieren, da der erste Operand von << kein `string` ist, sondern ein `ostream`.

Vertauschen

Das Vertauschen von Variablen ist ein grundlegender Bestandteil vieler Algorithmen. Realisiert wird es durch ein Prozedurtemplate aus der Standardbibliothek:

```
namespace std
{
    template <typename T>
    void swap(T& a, T& b)
    {
        T c(a);
        a = b;
        b = c;
    }
}
```

Leider ist `swap<string>` recht ineffizient, weil bei jedem Vertauschen eine dritte `string`-Variable `c` erzeugt wird und die internen Zeichenpuffer durch die beiden Zuweisungen überschrieben werden, bevor `c` wieder zerstört wird. Das Vertauschen von `string`s lässt sich durch Anbieten einer eigenen `swap`-Methode stark vereinfachen:

```
    void swap(string& that)
    {
        std::swap(buffer, that.buffer);
        std::swap(length, that.length);
    }
```

Das „tiefe Vertauschen“ ist nun gewissermaßen durch ein „flaches Vertauschen“ ersetzt worden. Damit diese Operation auch von den Standardalgorithmen verwendet wird, muss das Prozedurtemplate `swap` für `string` spezialisiert werden:

```
namespace std
{
    template <>
    void swap(::string& a, ::string& b)
    {
        a.swap(b);
    }
}
```

(Die Qualifikation `::string` ist hier lediglich notwendig, weil `string` im Namensraum `std` einen anderen Typ benennt.) Das Vertauschen zweier `string`-Variablen ist jetzt in konstanter Zeit durchführbar. Die Länge der enthaltenen Zeichenketten spielt keine Rolle mehr, und es findet garantiert keine Heap-Aktivität statt.

Transferkonstruktor

Die Initialisierung eines `string`s mit einem anderen `string` wird normalerweise vom Kopierkonstruktor vorgenommen. Falls der aktuelle Parameter ein rvalue ist, ist eine tiefe Kopie jedoch unnötig, da die Quelle kurz nach Anlegen der Kopie wieder zerstört wird. Stattdessen ist es deutlich effizienter, den Zeichenpuffer der Quelle zu „stehlen“:

```
string(string&& that)
{
    buffer = that.buffer;
    that.buffer = 0;
    length = that.length;
    that.length = 0;
}
```

Für Klienten bleibt die Veränderung von `that` unbemerkt, da das übergebene `string`-Exemplar nach der Rückkehr aus dem Transferkonstruktor nicht mehr ansprechbar ist, schließlich hat es auf Klientenseite keinen Namen (ansonsten würde `that` nicht an den aktuellen Parameter binden können, da `string&&` ausschließlich an rvalues bindet).

Transferzuweisung

Die Zuweisung vereinfacht sich ebenfalls erheblich, falls der rechte Operand ein rvalue ist. In diesem Fall kann einfach an `swap` delegiert werden, weil Zuweisung und Vertauschung zweier Variablen semantisch identisch sind, sofern die Quellvariable anschließend nicht mehr untersucht wird, was bei rvalues prinzipiell gegeben ist:

```
string& operator=(string&& that)
{
    swap(that);
    return *this;
}
```

Die Transferzuweisung wird beispielsweise bei `x = string("Banane")` aktiv, weil der Ausdruck `string("Banane")` ein rvalue ist. Abbildung 6.2 verdeutlicht den Prozess.

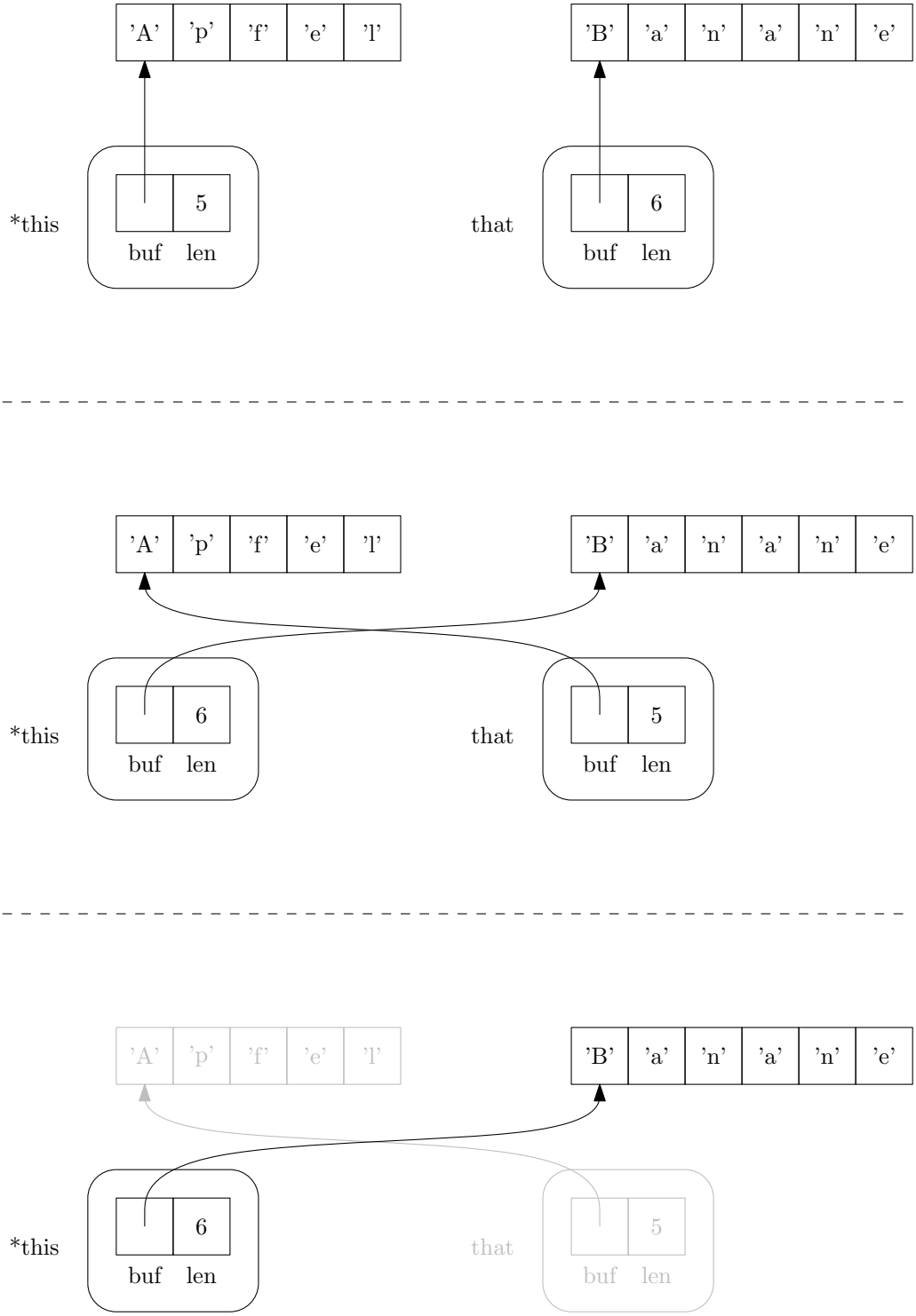

Abbildung 6.2: Transferzuweisung

Zusammenfassung

Die bisher erarbeitete Schnittstelle von `string` sieht wie folgt aus:

```
class string
{
    string(const char* p = "");              // Konstruktor

    // Korrektheit
    string(const string& that);              // Kopierkonstruktor
    string& operator=(const string& that);   // Kopierzuweisung
    ~string();                               // Destruktor

    // Effizienz
    void swap(string& that);                 // Vertauschen
    string(string&& that);                   // Transferkonstruktor (C++0x)
    string& operator=(string&& that);        // Transferzuweisung   (C++0x)

    // Ausgabe
    friend std::ostream& operator<<(std::ostream& os, const string& str);
};
```

Man beachte, dass noch keine einzige Wertoperation in der Schnittstelle auftaucht. Bisher wurde lediglich dafür gesorgt, dass der Umgang mit `string`-Exemplaren korrekt und effizient ist, und dass man sie auf Ausgabeströmen ausgeben kann.

Die Operationen zur Sicherstellung der Korrektheit sind *notwendig*, weil ein nativer Zeiger als Feld verwendet wird. Wenn man die Kopierzuweisung beispielsweise nicht von Hand definiert, dann würde bei einer Zuweisung `a = b` lediglich die enthaltene Länge und der Zeiger kopiert werden, nicht jedoch der referenzierte Zeichenpuffer. Der Zeiger `a.buffer` würde überschrieben werden, womit der alte Zeichenpuffer unwiederbringlich verloren ginge (*memory leak*). Nach der Zerstörung von `a` würde `b.buffer` auf einen nicht mehr existenten Zeichenpuffer verweisen (*dangling pointer*), und bei der Zerstörung von `b` würde der bereits zerstörte Zeichenpuffer erneut zerstört werden (*double delete*).

Die effizienzsteigernden Operationen sind *möglich*, weil `string` ein tiefer Werttyp ist. Hier ist das kontrollierte Umbiegen von Zeigern sinnvoll. Bei flachen Werttypen besteht dieses Optimierungspotential nicht, weil die Wertrepräsentationen dort vollständig im Exemplar selbst abgelegt sind. Daher wäre es beispielsweise sinnlos, `std::swap` für `point` zu spezialisieren. Weiterhin können die Transferoperationen bei flachen Werttypen nicht effizienter realisiert werden als die Kopieroperationen, weil dazu eine externe Wertrepräsentation notwendig ist, die ausgetauscht bzw. „gestohlen“ werden kann.

6.4.2 Automatische Speicherverwaltung

Die Implementation von `string` vereinfacht sich erheblich, wenn `char*` durch eine Klasse ersetzt wird, die selbstständig einen Puffer verwaltet. Ein passender Ersatz für `char*` ist `std::vector<char>`. Dieser wird in der Initialisierungsliste des Konstruktors befüllt:

```
class string
{
    std::vector<char> buffer;

public:

    string(const char* p = "") : buffer(p, p + strlen(p)) {}

//  1. Kopierkonstruktor
//  string(const string& that) : buffer(that.buffer) {}

//  2. Kopierzuweisung
//  string& operator=(const string& that)
//  {
//      buffer = that.buffer;
//      return *this;
//  }

//  3. Destruktor
//  ~string() {}

    void swap(string& that)
    {
        std::swap(buffer, that.buffer);
    }

//  4. Transferkonstruktor
//  string(string&& that) : buffer(std::move(that.buffer)) {}

//  5. Transferzuweisung
//  string& operator=(string&& that)
//  {
//      buffer = std::move(that.buffer);
//      return *this;
//  }

    // friend-Prozeduren können auch direkt im Klassenrumpf definiert werden
    friend std::ostream& operator<<(std::ostream& os, const string& str)
    {
        os.write(&str.buffer[0], str.buffer.size());
        return os;
    }
};
```

Man beachte, dass nun keinerlei manuelle Speicherverwaltung mehr erforderlich ist. Die Methoden, die normalerweise dafür zuständig sind, habe ich auskommentiert, da sie bei Bedarf automatisch vom Compiler generiert werden. Der Verzicht auf native Zeiger erleichtert die Implementation also enorm.

Die Transferoperationen delegieren an die entsprechenden Versionen in `std::vector`. Ohne Verwendung des Templates `std::move` würde stattdessen an die Kopieroperationen delegiert werden, weil der Ausdruck `that.buffer` ein lvalue ist. `std::move(expression)` behandelt `expression` als rvalue und ermöglicht damit einen Transfer.

Koenig Lookup und swap

Im Aufruf `std::swap(buffer, that.buffer)` ist die Qualifikation des Namensraums `std` eigentlich nicht notwendig, weil die Namensräume der aktuellen Parametertypen implizit berücksichtigt werden. Diese Sprachregel heißt *Argument Dependent Lookup* oder *Koenig Lookup*. Das Weglassen der Qualifikation ermöglicht die Verwendung von `swap`-Prozeduren, die sich nicht im Namensraum `std` befinden und ist daher besserer Stil.

Falls im Namensraum der aktuellen Parametertypen allerdings kein `swap` existiert, wäre `swap(a, b)` nicht übersetzbar. Damit notfalls auf `std::swap` zurückgegriffen werden kann, muss dieser Name mit einer `using`-Deklaration sichtbar gemacht werden:

```
void swap(string& that)
{
    using std::swap;
    swap(buffer, that.buffer);
}
```

Dieses Idiom funktioniert für alle Typen in allen Namensräumen. Dadurch ist die Implementation unabhängig von den Typen der Felder. Falls sich letztere ändern, muss `swap` nicht modifiziert werden, zum Beispiel beim Wechsel von nativern Zeigern auf Container.

6.4.3 Wertoperationen

Bisher bietet `string` keine Wertoperationen an. Dies soll im folgenden geändert werden.

Indizierung

Der Indizierungsoperator ermöglicht den Zugriff auf einzelne Zeichen mittels `s[i]`:

```
char operator[](size_t index) const
{
    if (index >= length()) throw std::out_of_range("illegal index");
    return buffer[index];
}

size_t length() const
{
    return buffer.size();
}
```

Der verwendete Index wird explizit auf Gültigkeit überprüft, um Programmierfehler auf Klientenseite abzufangen. `length()` liefert die Länge der Zeichenkette. `size_t` ist ein vorzeichenloser Ganzzahltyp für Größenangaben (oft definiert als `unsigned int`).

Vergleichsoperatoren

Die Vergleichsoperatoren lassen sich mit Standardalgorithmen implementieren:

```
    friend bool operator==(const string& x, const string& y)
    {
        return x.length() == y.length()
            && std::equal(x.buffer.begin(), x.buffer.end(), y.buffer.begin());
    }

    friend bool operator< (const string& x, const string& y)
    {
        return std::lexicographical_compare(x.buffer.begin(), x.buffer.end(),
                                            y.buffer.begin(), y.buffer.end());
    }
```

Konkatenation

Zum direkten Verändern eines `string`-Exemplars dient `operator+=`. Genau wie bei `point` wird `operator+` auf `operator+=` zurückgeführt:

```
    string& operator+=(const string& that)
    {
        buffer.insert(buffer.end(), that.buffer.begin(), that.buffer.end());
        return *this;
    }
};

string operator+(const string& x, const string& y)
{
    return std::move(string(x) += y);
}
```

Falls der linke Operand ein rvalue ist, kann das Exemplar direkt verändert werden:

```
string operator+(string&& x, const string& y)
{
    return std::move(x += y);
}
```

Interessanterweise unterscheiden sich die beiden Überladungen von `operator+` nur im ersten formalen Parametertyp und in der Frage, ob eine Kopie der Quelle stattfindet oder nicht. Dadurch lassen sich beide Überladungen vereinigen:

```
string operator+(string x, const string& y)
{
    return std::move(x += y);
}
```

Falls der linke Operand des Operators + ein lvalue ist, wird die notwendige Kopie implizit durch die Übergabe erzeugt. Ansonsten wird x mit dem Transferkonstruktor initialisiert, und es findet keine unnötige tiefe Kopie statt.

Teilstrings

Zunächst führe ich einen neuen Konstruktor ein, der einen string basierend auf einem std::vector<char> konstruiert:

```
string(std::vector<char> vec) : buffer(std::move(vec)) {}
```

Der formale Parametertyp basiert auf demselben Gedankengang wie bei operator+. lvalues bewirken den Aufruf des Kopierkonstruktors von std::vector, rvalues bewirken den Aufruf des Transferkonstruktors. Anschließend kann dieser neue string-Konstruktor beim Bilden von Teilstrings verwendet werden:

```
string substring(size_t start, size_t len)
{
    if (start + len > length())
        throw std::out_of_range("substring out of range");

    std::vector<char>::iterator pos = buffer.begin() + start;
    return string(std::vector<char>(pos, pos + len));
}
```

Bei rvalues kann das string-Exemplar direkt verändert werden:

```
string substring(size_t start, size_t len) &&
{
    if (start + len > length())
        throw std::out_of_range("substring out of range");

    std::vector<char>::iterator it = buffer.begin() + start;
    buffer.erase(pos + len, buffer.end());
    buffer.erase(buffer.begin(), pos);
    return *this;
}
```

6.4.4 Compilerkonformität

Die automatische Generierung von Transferkonstruktor und Transferzuweisung wird von Microsoft Visual C++ 2010 und g++ 4.5.0 noch nicht unterstützt. Momentan kann eine manuelle Implementation der Transferoperationen durch Delegation an die Transferoperationen der Felder also sinnvoll sein.

Die Überladung von Methoden anhand der Kategorie des Ausdrucks, der das aufzurufende Exemplar bezeichnet, wird von diesen beiden Compilern ebenfalls noch nicht unterstützt. Hier sind Optimierungen für rvalues also noch nicht umsetzbar, was ein weiterer Grund ist, so viele Operationen wie möglich als freistehende Funktionen zu realisieren.

6.4.5 Zusammenfassung

Tiefe Werttypen sollten keine nativen Zeigerfelder verwenden, weil die Sicherstellung der Korrektheit und Effizienz ansonsten recht komplex ist. Stattdessen sind Typen wie `std::vector<T>` vorzuziehen, weil die automatisch generierten Operationen dann bereits den gewünschten Effekt erzielen und nicht manuell geschrieben werden müssen.

Die beiden Überladungen `function(const T&)` und `function(T&&)` können durch eine einzige Version `function(T)` ersetzt werden. Je nachdem, ob der aktuelle Parameter ein lvalue oder ein rvalue ist, findet die Übergabe per Kopie oder per Transfer statt.

Falls der Ausdruck hinter dem `return` keine lokale Variable ist, muss dieser in ein `std::move` eingebettet werden, um den Transfer bei der Rückgabe zu ermöglichen.

6.5 Kochrezept für Werttypen in C++

Dieser Abschnitt fasst die Erkenntnisse aus den vorherigen Abschnitten zusammen und gibt komprimierte Hinweise zur Implementation und Verwendung benutzerdefinierter Werttypen in C++.

6.5.1 Verwendung

Kopiersemantik

In C++ sind Werttypen grundsätzlich mit Kopiersemantik zu verwenden. Ansonsten käme es schnell zu Lebenszyklus-Problemen und unerwünschten Seiteneffekten. Es gibt keinen Grund dafür, Wertexemplare dynamisch zu erzeugen.

Werttypen in Schnittstellen

Für die Übergabe von Werten kommen die formalen Parametertypen `T` und `const T&` in Frage. In den meisten Fällen ist `const T&` die bessere Wahl, lediglich bei primitiven Werttypen und einfachen flachen Werttypen ist `T` aus Effizienzgründen vorzuziehen, oder falls eine Kopie explizit erwünscht ist und ansonsten manuell stattfände. Die Rückgabe eines Wertes wird durch den Rückgabetyp `T` signalisiert.

6.5.2 Implementation

Felder

Als Felder kommen nur Werttypen und konkrete Typen in Frage. Letztere können dank *const correctness* bedenkenlos verwendet werden, weil Felder in Wertoperationen nicht verändert werden können. Um die Zuweisung auf Wertvariablen zu ermöglich, dürfen die Felder auf keinen Fall `const` sein:

```
class example
{
    std::string str_;
    int number_;
```

Konstruktoren

Die Felder werden in der Initialisierungsliste des Konstruktors initialisiert. Anschließend können Gültigkeitsprüfungen vorgenommen werden:

```
public:

    example(const std::string& str, int number)
    : str_(str), number_(number)
    {
        if (str_.empty()) throw std::domain_error("empty string");
        if (number_ < 0) throw std::domain_error("negative number");
    }
```

Falls der aktuelle Parameter ein rvalue ist, findet im Zuge der Initialisierung eine unnötige Kopie statt. In C++0x kann man diese unnötige Kopie verhindern, indem man die Übergabe per Konstruktion wählt und anschließend explizit transferiert:

```
public:

    example(std::string str, int number)
    : str_(std::move(str)), number_(number)
    {
        if (str_.empty()) throw std::domain_error("empty string");
        if (number_ < 0) throw std::domain_error("negative input");
    }
```

Hier ist besonders darauf zu achten, im Konstruktorrumpf `str_` statt `str` zu verwenden, weil `str` nach der Initialisierung von `str_` prinzipbedingt leer ist – der Transferkonstruktor hat die Ressourcen von `str` bereits an `str_` übertragen.

Coercion

Konstruktoren einer Klasse `Z`, die mit einem Parameter vom Typ `Q` aufrufbar sind, definieren eine Coercion von `Q` nach `Z`. Falls das unerwünscht ist, sollten diese Konstruktoren mit `explicit` markiert werden. Die Methode `Q::operator Z() const` definiert ebenfalls eine solche Coercion, welche allerdings erst in C++0x deaktivierbar ist.

Allgemein sollte man mit Coercion eher sparmsam umgehen, da unerwartete implizite Typumwandlungen nicht zur Verständlichkeit von Quellcode beitragen.

swap

Die `swap`-Methode kann bei tiefen Werttypen einen Performancegewinn bringen:

```
void swap(example& that)
{
    using std::swap;
    swap(str_, that.str_);
    swap(number_, that.number_);
}
```

Weiterhin ist eine freistehende `swap`-Prozedur zu implementieren, die an die `swap`-Methode delegiert. Am sichersten ist die Spezialisierung von `std::swap`, weil nicht alle Programmierer das `using`-Idiom beherrschen:

```
namespace std
{
    template <>
    void swap(example& a, example& b)
    {
        a.swap(b);
    }
}
```

Der Rückgabetyp `void` stellt übrigens sicher, dass `swap` nicht versehentlich als vermeintliche Wertoperation verwendet wird, da es keine Teilausdrücke vom Typ `void` gibt.

Zusammengesetzte Zuweisungen

Genau wie die normale Zuweisung verändern zusammengesetzte Zuweisungen die Felder. Daher können diese nicht als `const` markiert werden, wodurch sie sich zusätzlich von den Wertoperationen abheben:

```
    example& operator+=(const example& that)
    {
        str_ += that.str_;
        number_ += that.number_;
        return *this;
    }
```

Der Rückgabetyp ist `T&`, weil konzeptionell kein Wert zurückgegeben wird, sondern die veränderte Variable. Da zusammengesetzte Zuweisungen lediglich als Optimierung zu verstehen sind, muss jeweils auch das funktionale Gegenstück ohne `=` angeboten werden.

Wertoperationen

Wertoperationen sind vorzugsweise als freistehende Funktionen zu realisieren. `x = x + y` sollte dieselbe Semantik wie `x += y` haben. `operator+` kann auf `operator+=` zurückgeführt werden, um Code-Redundanz zu vermeiden:

```
example operator+(example x, const example& y)
{
    return x += y;
}
```

Falls hinter dem `return` ein lvalue steht, sollte in C++0x bei tiefen Werttypen `std::move` verwendet werden, um die Rückgabe per Transfer zu ermöglichen:

```
example operator+(example x, const example& y)
{
    return std::move(x += y);
}
```

Innerhalb von Wertklassen sind Wertoperationen grundsätzlich als `const` zu markieren, damit sie auch auf Konstanten aufrufbar sind:

```
int number() const
{
    return number_;
}
```

Bei der Rückgabe von Feldern aus Methoden kann alternativ der Rückgabetyp `const T&` verwendet werden, um eine teure Kopie zu sparen:

```
const std::string& string() const
{
    return str_;
}
```

Hierbei müssen Klienten allerdings eigenständig darauf achten, die zurückgegebene Referenz nicht längerfristig zu speichern.

Prädikate

Der Test auf Gleichheit wird durch Überladen von `operator==` realisiert:

```
bool operator==(const example& x, const example& y)
{
    return (x.string() == y.string())
        && (x.number() == y.number());
}
```

Um die Verwendung des Werttyps in `set` und `map` zu ermöglichen, muss `operator<` überladen werden. Nicht jeder Werttyp hat jedoch eine natürlich Ordnung. In diesem Fall muss eine künstliche Ordnung eingeführt werden:

```
bool operator<(const example& x, const example& y)
{
    if (x.string() < y.string()) return true;
    if (y.string() < x.string()) return false;

    return x.number() < y.number();
}
```

Alle anderen Vergleichsoperatoren können auf == und < zurückgeführt werden:

```
bool operator!=(const example& x, const example& y) { return !(x == y); }

bool operator> (const example& x, const example& y) { return y < x;     }

bool operator>=(const example& x, const example& y) { return !(x < y);  }

bool operator<=(const example& x, const example& y) { return !(y < x);  }
```

In Abschnitt 6.6.3 werde ich zeigen, wie diese redundanten Operatoren automatisch generiert werden können, um mühselige Schreibarbeit zu sparen.

6.6 Evaluation: C++ aus der Werttyp-Perspektive

In diesem Abschnitt möchte ich drei Fragen zur Unterstützung von Werttypen in C++ beantworten: Wie einfach ist die Benutzung durch Klienten? Welche Sprachmechanismen unterstützen das Wertmodell? Wieviel Schreibaufwand ist notwendig?

6.6.1 Benutzung

Aus Klientensicht ist der Umgang mit zusammengesetzten Werttypen sehr einfach. Sie fühlen sich im Großen und Ganzen wie primitive Werttypen an. Es ist keine Zeigersemantik notwendig, wodurch keine Probleme mit Lebenszyklen auftreten. Um eine versehentliche dynamische Erzeugung zu verhindern, kann man `operator new` innerhalb der Wertklasse verbergen:

```
class point
{
    // ...
private:
    void* operator new(size_t bytes);
};
```

Dann führt `new point(3, 4)` zu einem Übersetzungsfehler. `operator new` kümmert sich ausschließlich um die dynamische Speicherverwaltung und ist nicht mit dem Konstruktor zu verwechseln. Variablen, Konstanten und temporäre Exemplare können nach wie vor konstruiert werden. Um die dynamische Erzeugung nicht in jeder Wertklasse erneut verbieten zu müssen, kann man eine simple Basisklasse definieren, von der man erbt:

```
class not_newable
{
private:
    void* operator new(size_t);
};

class point : not_newable
{
    // ...
};
```

Den Adressoperator sollte man nicht verbergen, da die Adressbildung von Variablen in einigen Kontexten sinnvoll sein kann, zum Beispiel beim Sortieren von Arrays:

```
std::string array[] = {"hello", "beautiful", "world"};
std::sort(&array[0], &array[3]);
```

Ein Verbergen von `operator&` würde hier zu einem Übersetzungsfehler führen. Bei den Anforderungen an Typen, die in Sammlungen verwendet werden, wird außerdem explizit gefordert, dass `&t` die Adresse von `t` bezeichnen muss [ISO03, 20.1.3]. Überladungen von `operator&` sind daher nicht zu empfehlen. Dies ist aber im Zusammenhang mit Werttypen nicht weiter problematisch, da die Adressbildung auf rvalues sowieso nicht erlaubt ist.

Saubere Gleichheit

Zum Vergleich von Werten können, bei entsprechender Definition des Werttyps, die gewohnten Infix-Operatoren verwendet werden. Es gibt darüber hinaus keine weiteren, semantisch abweichenden Vergleichsoperationen.

Inkompatible Werte können nicht versehentlich miteinander verglichen werden. Ein solcher Versuch wird mit einem Übersetzungsfehler quittiert. In der Implementation sind dazu weder Typprüfungen noch Typzusicherungen erforderlich.

Keine `null`-Probleme

Da Werttypen in C++ mit Kopiersemantik verwendet werden, kann zur Laufzeit keine `NullPointerException` oder ähnliches auftreten. Auch bei der Übergabe per konstanter Referenz kann es zu keinen `null`-Problemen kommen, da in wohlgeformten Programmen keine `null`-Referenzen existieren [ISO03, 8.3.2 dcl.ref §5].

Potentiell ungebundene Variablen können durch `boost::optional<T>` realisiert werden.

Effizienz

Einfache zusammengesetzte Werttypen wie `point` sind äußerst effizient. Ein Array von `n` Punkten benötigt nicht mehr Speicher als zwei Arrays von `n` Ganzzahlen. Es gibt keine zusätzlichen Metadaten oder Indirektionen. Man muss als Klient keine Performanceeinbußen fürchten, wenn man solche einfachen zusammengesetzten Werttypen verwendet.

Aufgrund der Kopiersemantik sind die Wertrepräsentationen nicht gemeinsam nutzbar. Bei tiefen Werttypen wie `string` ist die Zuweisung dadurch sehr teuer, weil die komplette Zeichenkette kopiert werden muss. Das Idiom *copy on write* zögert diesen Kopiervorgang in der Hoffnung hinaus, dass dieser in vielen Fällen unnötig ist, zum Beispiel weil die Quellvariable kurz nach der Zuweisung verschwinden wird. Eine korrekte Implementation von *copy on write* ist jedoch sehr komplex, und bei Multithreading treten wegen der erforderlichen Synchronisierung signifikante Performanceprobleme auf [Sut01, Item 16].

In C++0x werden Kopieroperationen in vielen Kontexten durch Transferoperationen abgelöst, weshalb die Kopierproblematik bei tiefen Werttypen an Gewicht verliert.

Parameterübergabe

Zusammengesetzte Werttypen sollten sich wie primitive Werttypen anfühlen. Letztere werden durchgehend per Kopie übergeben, weil die Übergabe per konstanter Referenz keinen Vorteil hätte. Bei zusammengesetzten Werttypen hat der Programmierer dagegen die Wahl zwischen Übergabe per Konstruktion und Übergabe per konstanter Referenz. Die Schnittstellen werden dadurch mit semantisch irrelevanten Effizienzdetails belastet.

In C++0x hängt die Wahl der Konstruktorparameter außerdem davon ab, ob die verwendeten Typen effizient transferierbar sind.

6.6.2 Unterstützende Sprachmechanismen

Die Unveränderlichkeit wird durch *const correctness* unterstützt, unter anderem weil in konstanten Methoden ein versehentliches Ändern der Felder unmöglich ist. Für die Seiteneffektfreiheit und referenzielle Transparenz gibt es in C++03 keine Unterstützung.

C++0x führt ein neues reserviertes Wort `constexpr` ein, welches die Auswertung von Funktionsaufrufen zur Übersetzungszeit ermöglicht. Der Rumpf einer solchen Funktion muss aus einer einzigen `return`-Anweisung bestehen, und es können keine Zustände ausgelesen oder verändert werden. Da es in `constexpr` Funktionen keine lokalen Variablen gibt, müssen imperative Algorithmen durch rekursive ersetzt werden. Die Wurzelfunktion aus dem Grundlagenkapitel könnte beispielsweise wie folgt implementiert werden:

```
constexpr double wurzel_rekursiv(double x, double guess, double old_guess)
{
    return (guess == old_guess) ? guess
        : wurzel_rekursiv(x, (guess + x/guess) * 0.5, guess);
}

constexpr double wurzel(double x)
{
    return wurzel_rekursiv(x, 1.0, x);
}
```

Mangels Compilerunterstützung konnte ich bisher leider keine praktischen Erfahrungen mit diesem Sprachmechanismus sammeln.

6.6.3 Schreibaufwand

Bei einfachen zusammengesetzten Werttypen wie `point` ist der Schreibaufwand genau wie in anderen Sprachen recht hoch, da der Programmierer den technischen Konstruktor, die Getter und die Vergleichsoperatoren von Hand schreiben muss. Änderungen im Werttyp ziehen daher fehleranfällige Wartungsarbeit nach sich.

Redundante Operatoren

Viele Operatoren sind semantisch redundant und können auf bestehende Operatoren zurückgeführt werden. Um diese redundanten Operatoren nicht für jede Wertklasse neu definieren zu müssen, kann man ein Klassentemplate schreiben, das passende `friend`-Funktionen definiert. Durch Erben von diesem Klassentemplate, parametrisiert mit dem eigenen Werttyp, werden die redundanten Operatoren bei Bedarf automatisch erzeugt:

```
template <typename T>
class addable
{
    friend T operator+(T x, const T& y)
    {
        return std::move(x += y);
    }
};
```

```
template <typename T>
class comparable
{
    friend bool operator!=(const T& x, const T& y)
    {
        return !(x == y);
    }

    friend bool operator> (const T& x, const T& y)
    {
        return y < x;
    }

    friend bool operator>=(const T& x, const T& y)
    {
        return !(x < y);
    }

    friend bool operator<=(const T& x, const T& y)
    {
        return !(y < x);
    }
};

class point : not_newable, addable<point>, comparable<point>
{
    // ...

    point& operator+=(const point& y);             // notwendig für addable
};

bool operator==(const point& x, const point& y);   // notwendig für comparable

bool operator< (const point& x, const point& y);   // notwendig für comparable
```

Hierbei findet übrigens keine Vererbung im eigentlichen Sinn statt, weil `comparable` und `addable` weder Felder noch Methoden definieren. Durch den privaten Vererbungsmodus gibt es auch keine Subtypbeziehung zwischen `point` und `addable<point>` bzw. `comparable<point>`. Dieser Trick wurde von Barton und Nackmann entdeckt [VJ02, 11.7] und in `boost::operators` perfektioniert.

6.6.4 Gesamteindruck

Die Implementation von Werttypen ist in C++ recht komplex. Abgesehen von dem Schreibaufwand, der auch in anderen Sprachen erforderlich ist, muss der Programmierer sich unter anderem mit Ausdruckskategorien, *const correctness* und drei bzw. vier verschiedenen Parameterübergabemechanismen auseinandersetzen.

Das Verhalten von Variablen sollte bei der Implementation eines Werttyps eigentlich gar keine Rolle spielen. Die fehlende Trennung von Variablen und Exemplaren verhindert eine saubere Umsetzung des Wertmodells in C++.

Die Benutzung von Werttypen ist in C++ verhältnismäßig einfach. Insbesondere gibt es beim Vergleich keine unerwarteten Stolpersteine.

6.7 Evaluation: Werttypen aus der C++ Perspektive

Bisher wurde untersucht, wie Werttypen in C++ umgesetzt werden können. In diesem Abschnitt möchte ich kurz diskutieren, wie sinnvoll Werttypen aus der Perspektive eines C++ Programmierers erscheinen.

6.7.1 Zeichenketten

In Abschnitt 6.1.2 wurde bereits besprochen, dass `std::string` veränderliche Zeichenketten modelliert, weshalb es sich bei diesem Typ um keinen Werttyp handelt. Die verändernden Operationen wirken sich allerdings auf derselben Ebene aus wie die Zuweisung, d.h. es sind niemals mehrere `string`-Variablen gleichzeitig betroffen. Beispielsweise haben die folgenden Anweisungspaare jeweils dieselben Seiteneffekte:

```
a = b;          // a wird an denselben string gebunden wie b
a.assign(b);    // a wird an denselben string gebunden wie b

a += b;         // a wird an a + b gebunden
a.append(b);    // a wird an a + b gebunden
```

Die garantierte Unveränderlichkeit von Werten im Sprachmodell ist in C++ nicht erforderlich, da verändernde Operationen keine Werte verändern, sondern Variablen umbinden. Die gefürchteten impliziten Aliasing-Effekte existieren in C++ nicht. Es bringt keine zusätzliche Sicherheit, verändernde Operationen zu verbieten.

6.7.2 Funktionale Datenstrukturen

Wenn man Werttypen durch unveränderliche Exemplare realisiert, auf die gemeinsam zugegriffen wird, dann lassen sich bestimmte Operationen sehr effizient realisieren. Das Einfügen eines Elements am Anfang einer einfach verketteten, unveränderlichen Liste ist beispielsweise in konstanter Zeit durchführbar, weil nur ein einziger Knoten angelegt werden muss und die Restliste wiederverwendet werden kann. Unveränderliche Listen, Mengen und Abbildungen werden beispielsweise in den Standardbibliotheken von Scala [OSV08] und Clojure [Hal09] mitgeliefert.

In C++ sind solche Datenstrukturen mangels Garbage Collector nicht effizient realisierbar. Daher verwendet man in C++ bevorzugt veränderliche Listen, bei denen das Einfügen am Ende einer Liste in amortisiert konstanter Zeit durchführbar ist. Wenn man eine wertartige Liste in C++ implementieren würde, ohne gemeinsame Teilstrukturen wiederverwenden zu können, dann würde jedes Einfügen (und jede Zuweisung) lineare Zeit beanspruchen. Für den praktischen Einsatz wären solche Typen viel zu ineffizient.

6.8 Fazit

C++ unterstützt Werte ausschließlich in dem Sinn, dass auf Ausdrucksebene zwischen Variablen und den an sie gebundenen Werten unterschieden werden kann. Diese Unterscheidung ist jedoch sehr sprachspezifisch und aufgrund einiger Sonderregeln nicht leicht nachvollziehbar. [1] Die Regeln der Parameterübergabe berücksichtigen diesen Unterschied, d.h. die Signaturen geben Auskunft darüber, ob Werte oder Variablen verarbeitet werden.

Das Problem des versehentlichen gemeinsamen Zugriffs auf Exemplare existiert aufgrund der Kopiersemantik nicht. Aus diesem technischen Detail ist ein Programmierstil entstanden, der imperative und funktionale Konzepte vermischt. Variablen können nicht nur per Zuweisung an andere Werte gebunden werden, sondern durch beliebige verändernde Operationen, die anhand ihrer Signatur immerhin eindeutig als solche erkennbar sind. Der durchgehend funktionale Stil, der ein zentraler Bestandteil des Wertmodells ist, verträgt sich nicht gut mit der Kopiersemantik von C++, was spätestens bei den funktionalen Datenstrukturen offensichtlich wird.

Bei der Implementation von Wertklassen werden veränderliche Variablen modelliert, die an Werte gebunden sind, und nicht die Werte selbst. Das eigentliche Ziel von Werttypen im Sinne dieser Arbeit, nämlich eine saubere Modellierung von Werten und ihren Beziehungen untereinander, ist in C++ somit nicht realisierbar.

[1] Die Unterscheidung zwischen *lvalues* und *rvalues* wurde bisher in keinem C++ Buch, das ich gelesen habe, korrekt erklärt. Hinzu kommt, dass die Begriffe in anderen Programmiersprachen wie C oder BCPL abweichende Bedeutungen haben und manchmal sogar sprachunabhängig verwendet werden. Im C99-Standard wurde der Begriff *rvalue* ersatzlos gestrichen. Im C++0x-Standard gibt es neben *lvalues* und *rvalues* drei neue Ausdruckskategorien: *xvalues*, *glvalues* und *prvalues*.

7 Zusammenfassung

In dieser Arbeit sollte untersucht werden, wie benutzerdefinierte Werttypen in C++ umgesetzt werden können. Dazu wurde zunächst das von Beate Ritterbach et al. herausgearbeitete Wertmodell zusammengefasst. Anschließend wurden die bisherigen Erkenntnisse zur Umsetzung des Wertmodells in Java diskutiert. Dabei wurde deutlich, dass eine Sprachunterstützung für Werttypen grundsätzlich sinnvoll erscheint.

Im Zuge einer Klärung der für diese Arbeit fundamentalen Sprachmechanismen von C++ erfolgten einige Umbenennungen, um Kollisionen mit Begriffen des Wertmodells zu vermeiden. Im Hauptteil wurde dann untersucht, wie das Wertmodell in C++ umgesetzt werden kann. Dabei stellte sich heraus, dass der ungewöhnliche Objektbegriff von C++ und insbesondere die fehlende Trennung zwischen Variablen und Exemplaren eine saubere Umsetzung des Wertmodells in C++ massiv erschweren.

Weiterhin wurde deutlich, dass die Zuweisung bei Verwendung von Kopiersemantik nicht effizient durchführbar ist, weshalb komplexere Werttypen wie beispielsweise unveränderliche Listen in C++ inakzeptable Komplexitätsgarantien hätten. Eine einfache Spracherweiterung in Anlehnung an VJ, die „ValueC++“ in normales C++ übersetzt, erscheint unter diesen Umständen wenig sinnvoll. Ohne Referenzsemantik und Garbage Collection sind Werttypen im Sinne dieser Arbeit nicht effizient realisierbar.

7.1 Ausblick

Sofern Garbage Collection in einen zukünftigen C++ Standard aufgenommen werden sollte, könnte man das Thema *Benutzerdefinierte Werttypen in C++* neu aufrollen. Ein technisches Problem würde dann beispielsweise der Infix-Vergleich `a == b` darstellen, da die Semantik des Zeigervergleichs in C++ nicht redefinierbar ist. (Bei benutzerdefinierten Vergleichen muss mindestens ein benutzerdefinierter Typ involviert sein, und Zeiger sind keine benutzerdefinierten Typen.)

Weiterhin wäre es erforderlich, genügend C++ Programmierer von dem Wertmodell zu überzeugen. In einem 2007 an der Universität Waterloo aufgezeichneten Video sagt Bjarne Stroustrup über den Standardisierungsprozess von C++:

> We cannot even take all the „good“ proposals. Define a good proposal as something that will make life easier for 200.000 people. That's not sufficient. If we took all of those, we would not be able to handle the results.

Hier wäre also noch viel Überzeugungsarbeit zu leisten, bevor ein Werttyp-Vorschlag realistische Chancen hätte, vom Standardisierungskomitee akzeptiert zu werden.

Literaturverzeichnis

[Bec06] Pete Becker. *The C++ Standard Library Extensions: A Tutorial and Reference.* Addison-Wesley, 2006.

[Blo08] Joshua Bloch. *Effective Java.* Addison-Wesley, 2nd edition, 2008.

[Car94] Tom Cargill. Exception Handling: A False Sense Of Security. *C++ Report*, 6, 1994.

[FS02] Martin Fürther, Annika Spreckelmeyer. Konzepte und Ansätze zur Unterstützung der Implementierung fachlicher Werte in objektorientierten Programmiersprachen am Beispiel des Java-Rahmenwerks JWAM. Diplomarbeit, 2002.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, 3rd edition, 2005.

[Hal09] Stuart Halloway. *Programming Clojure.* Pragmatic Programmers, 2009.

[Hei05] Markus Heiden. Generierung von Fachwerten aus einem abstrakten Sprachmodell. Diplomarbeit, 2005.

[ISO03] ISO/IEC 14882. *The C++ Standard, Incorporating Technical Corrigendum 1*, 2003.

[Jos99] Nicolai Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, 1999.

[Kar05] Björn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost.* Addison-Wesley, 2005.

[KM00] Andrew Koenig and Barbara Moo. *Accelerated C++.* Addison-Wesley, 2000.

[LML05] Stanley Lippman, Barbara Moo, and Josee Lajoie. *C++ Primer.* Addison-Wesley, fourth edition, 2005.

[Mac82] B. J. MacLennan. Values and Objects in Programming Languages. Technical report, Naval Postgraduate School, 1982.

[Mül99] Klaus Müller. Konzeption und Umsetzung eines Fachwertkonzepts. Studienarbeit, 1999.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *The Scala Programming Language.* artima, 2008.

[Rat06] Jörg Rathlev. Ein Werttyp-Konstruktor für Java. Diplomarbeit, 2006.

[Sau01] Joachim Sauer. Konfiguration von Fachwerten mit XML-Definitionen. Studienarbeit, 2001.

[Sch94] Axel-Tobias Schreiner. *Objektorientierte Programmierung mit ANSI C*. Hanser Fachbuch, 1994.

[Sch07a] Axel Schmolitzky. Leave out the modelling when teaching object-orientation to beginners. Technical report, 2007.

[Sch07b] Axel Schmolitzky. Sieben Thesen zur erfolgreichen Verwirrung von Anfängern der objektorientierten Programmierung. *Informatik Spektrum*, 30:1, 2007.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[Str08] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Addison-Wesley, 2008.

[Sut01] Herb Sutter. *More Exceptional C++*. Addison-Wesley, 2001.

[VJ02] David Vandevoorde and Nicolai Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

[Wal08] Marek Walczak. IDE-Unterstützung für graduelle und individuelle Sprachererweiterung. Diplomarbeit, 2008.

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.