22

Performance, Wert- und Referenzsemantik

Dieses Kapitel behandelt die folgenden Themen:

- Performanceproblem Wertsemantik
- Optimierung durch Referenzsemantik für R-Werte
- Vermeidung temporärer Objekte

Von Wertsemantik spricht man, wenn auf Objekte direkt zugegriffen wird. Dem gegenüber steht die Referenzsemantik, die einen Zugriff auf Objekte nur über Verweise oder Referenzen zulässt. Java verwendet bei Klassenobjekten Referenzsemantik, C++ baut auf der Wertsemantik auf. In C++ lässt sich eine Referenzsemantik gut mit Zeigern oder Referenzen realisieren. Die Konsequenzen seien anhand einiger Beispiele gezeigt, wobei die Existenz der Klasse `A` gegeben sein soll:

```
A a1;
A a2 = a1;            // Wertsemantik: Eine echte Kopie wird erzeugt
A* aptr1 = new A;
A* aptr2 = aptr1;     // Referenzsemantik: nicht das Objekt, sondern
                              // nur der Verweis/Zeiger wird kopiert.
if(a1 == a2) ...      // Wertsemantik: Prüfung auf Gleichheit
if(aptr1 == aptr2) ... // Referenzsemantik: Prüfung auf Identität
```

Wenn zwei Zeiger gleich, aber nicht Null sind, verweisen sie auf dasselbe Objekt. Wenn zwei Zeiger ungleich sind, verweisen sie auf verschiedene Objekte, die untereinander gleich oder ungleich sein können. Die Tabelle 22.1 zeigt eine Übersicht, wobei mit Referenzen auch Zeiger gemeint sind.

Tabelle 22.1: Wert- und Referenzsemantik

Unterschied bezüglich	Wertsemantik	Referenzsemantik
Zugriff	direkt	über Referenz
`==`	prüft Gleichheit	vergleicht Identität
`=`	kopiert Objekt	kopiert Referenz
Sichtbarkeitsregeln	für Objekte	für Referenzen
Lebensdauer	Objekt existiert nur innerhalb des Gültigkeitsbereichs	Objekt existiert unabhängig von der Referenz
Speicherort für Objekte	Stack	Heap
Parameterübergabe	`f(A a)`	`f(const A& a)` `f(const A* ptr)`
Wertrückgabe	`A g()`	`A& g()`, `A* g()` (semantisch nicht dasselbe)

Besonders beim Einsatz von Containern sollte man sich genau überlegen, ob echte Kopien (Wertsemantik) gewünscht sind oder nicht:

```
// Vektor mit (möglicherweise aufwendigen) Kopien
vector<A> v1(10000);

// Initialisierung aller Elemente (weggelassen)

vector<A> v2 = v1;       // 10000 Aufrufe von A(const A&).
v2[1].setAttribut(100); // wirkt sich nicht auf v1[1] aus.

// Vektor mit Zeigern
vector<A*> vp1(10000);

// Initialisierung aller Elemente (weggelassen)

vector<A*> vp2 = vp1;    // Kein Aufruf von A(const A&).
v2[1]->setAttribut(100); // wirkt sich auf v1[1] aus.
```

Wenn keine Kopien gewünscht sind, empfiehlt es sich, mit Zeigern zu arbeiten. Das kann bei großen Containern einen erheblichen Performance-Gewinn bedeuten. Dabei ist natürlich zu beachten, dass mit `new` angelegte Objekte nach Verwendung irgendwann mit `delete` zerstört werden müssen. Wenn man sich darum nicht kümmern möchte, sollte man im Vektor die Zeiger als `shared_ptr` ablegen:

```
for(int i=0; i < vp1.size(); ++i) {
   vp1[i] = std::shared_ptr<A>(new A(i));
}
```

Informationen zu `shared_ptr` finden Sie auf den Seiten 344 und 851.

22.1 Performanceproblem Wertsemantik

Die Wertsemantik führt manchmal zu überflüssigen Kopien. Sehen wir uns dazu einen +-Operator an, der zwei Strings verkettet und dabei den als vorhanden angenommenen +=-Operator nutzt:

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

Bei der Parameterübergabe wird der Aufruf des Kopierkonstruktors durch die Übergabe von Referenzen vermieden. Ohne weitere Optimierung wird er aber für `StringTyp(a)` aufgerufen und noch einmal bei der Rückgabe des Ergebnisses. Der Compiler kann entsprechend der Bemerkung auf Seite 159 den zweiten Aufruf eliminieren (englisch *named return value optimization (NRVO* oder kurz *RVO)*). Dies kann leicht gezeigt werden, wenn der Kopierkonstruktor mit einer protokollierenden Ausgabe wie etwa

```
std::cout << "Kopierkonstruktor gerufen" << std::endl;
```

versehen wird. Die zwei Fälle (mit und ohne Optimierung) können mit dem GNU C++-Compiler verglichen werden, indem die Übersetzung ohne beziehungsweise mit dem Parameter `-fno-elide-constructors` durchgeführt wird.

22.1.1 Auslassen der Kopie

Das Auslassen der Kopie (englisch *copy elision*) durch den Compiler steigert die Performance. Das führt auf eine weitere Optimierungsmöglichkeit, wenn temporäre Objekte beteiligt sind. Sehen Sie sich dazu die folgenden beiden Varianten an (nach [Abr]):

```
// Auszug aus cppbuch/k22/RVO/main.cpp
StringTyp str1("Erster");
StringTyp str2("Zweiter");
StringTyp erg1(str1 + str2);                  // 1) siehe unten
StringTyp erg2(StringTyp("temporaer") + str2); // 2) siehe unten
```

`str1` und `str2` sind benannte Variablen, also eindeutig L-Werte. `StringTyp("temporaer")` hingegen ist ein R-Wert – er kann nicht auf der linken Seite einer Zuweisung stehen. Betrachten Sie nun die Konsequenzen bei Aufruf des obigen +-Operators:

1. `StringTyp erg1(str1 + str2);`
   ohne RVO: Der obige +-Operator hat `a` als ersten Parameter. Davon erzeugt er eine Kopie (1. Aufruf des Kopierkonstruktors mit `"Erster"`). An diese Kopie wird `b` angehängt. Das Ergebnis `"ErsterZweiter"` wird per `return` zurückgegeben (2. Aufruf des Kopierkonstruktors mit `"ErsterZweiter"`). Die Bildung von `erg1` erfordert schließlich den 3. Aufruf des Kopierkonstruktors.
   mit RVO: Der Aufruf des Kopierkonstruktors bei der Rückgabe wird eingespart.
2. `StringTyp erg2(StringTyp("temporaer") + str2);`
   ohne RVO: Verhalten wie unter 1. beschrieben.

mit RVO: Verhalten wie unter 1. beschrieben. Aber: Eine genaue Betrachtung zeigt, dass das temporäre Objekt `StringTyp("temporaer")` nach Aufruf der Operatorfunktion nicht weiter benötigt wird. Der erste Kopierkonstruktor wäre überflüssig, wenn dieses temporäre Objekt direkt an die Stelle von `a` treten könnte. Man müsste nur dem Compiler die Möglichkeit geben, RVO einzusetzen.

Dazu wird der erste Parameter einfach *per Wert* übergeben, und auf die explizite Kopie wird verzichtet.

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

Der Compiler hat nun die Möglichkeit, das temporäre Objekt direkt an der Stelle von `a` zu verwenden. Ergebnis: Es ist nur *ein einziger* Aufruf des Kopierkonstruktors notwendig, nämlich der zur Erzeugung von `erg2`!

Wir haben hier das auf den ersten Blick paradox erscheinende Verhalten, dass die Übergabe per Wert performanter als die Übergabe per Referenz sein kann. Falls der erste Parameter ein nicht temporäres Objekt ist, wird der Kopierkonstruktor vom Compiler ganz normal aufgerufen. Daraus ergibt sich die folgende

Empfehlung

Falls in einer Funktion die lokale Kopie eines Arguments benötigt wird (und nur dann!) sollte das Argument per Wert übergeben werden.

Dann hat der Compiler die Möglichkeit, mit RVO den Kopierkonstruktor für temporäre Objekte einzusparen. Im Fall des obigen `+`-Operators gilt dies nur für das Argument `a`, nicht für `b`.

22.1.2 Temporäre Objekte bei der Zuweisung

Die oben beschriebene Problematik gilt auch in anderen Zusammenhängen. Beim Aufruf

```
ergstr = stra + strb; // Alle Variablen sind vom Typ StringTyp.
```

wird ohne RVO auf der rechten Seite der Kopierkonstruktor einmal bei der Bildung des temporären Objekts aufgerufen, dann ein weiteres Mal bei der Rückgabe und ein drittes Mal beim Aufruf des Zuweisungsoperators, der dem bekannten Muster aus Abschnitt 9.2.2 folgt:

```
// Auszug aus cppbuch/k22/RVO/einfacherstring.h
StringTyp& operator=(StringTyp s) { // Zuweisungsoperator
  swap(s);
  return *this;
}
```

Der `StringTyp`-Zuweisungsoperator übernimmt den Parameter per Wert (= Aufruf des Kopierkonstruktors) und vertauscht die Kopie mit `*this`. Danach gibt der Destruktor den alten Wert der linken Seite der Zuweisung, der nun in `s` steckt, frei. Das Überladen von Operatoren hat einen Preis, wie man sieht. *Mit* RVO wird wie oben ein Aufruf weniger

gebraucht. Der C++-Standard schreibt die Optimierung nicht vor; sie wird dem Compiler überlassen. Ein semantisch äquivalenter Funktionsaufruf etwa der Art

```
ergstr.verketten(stra, strb);
```

benötigt zwar eine `new`-Operation, wenn der aufnehmende String keine Platzreseve hat, würde aber gänzlich ohne temporäre Objekte und Aufruf des Kopierkonstruktors auskommen. Aus diesem Grund hat sich das C++-Standard-Komitee eine Spracherweiterung einfallen lassen, die im nächsten Abschnitt beschrieben wird.

22.2 Optimierung durch Referenzsemantik für R-Werte

Das Problem ist klar zu identifizieren. Es ist die Wertsemantik, die zur Kopie von Werten auch in den Fällen führt, in denen die Weiterleitung einer Referenz genügen würde.

- Die Funktion `operator+(StringTyp, const StringTyp&)` gibt ein Ergebnis zurück, das temporär ist und nach Ende der Funktion ohnehin verschwindet. Mit anderen Worten, es könnte direkt auf der rechten Seite einer Zuweisung eingesetzt werden, anstatt den Kopierkonstruktor aufzurufen.
- Ähnliches gilt für die Zuweisung selbst. Statt die rechte Seite der Zuweisung zu kopieren, könnte die rechte Seite direkt an die Stelle von `ergstr` treten. Das geht natürlich nur, wenn rechts wirklich ein temporäres Objekt auftritt, das nach Vollendung der Zuweisung verschwindet. Eine Zeile wie `ergstr = stra;` erfordert jedoch eine echte Kopie, weil beide Variablen danach bestehen bleiben.

Es genügt also in bestimmten Fällen, wenn Objekte *bewegt* anstatt *kopiert* werden (englisch *move semantics* bzw. *copy semantics* oder *value semantics*). Die Frage ist, wie ein Compiler klar erkennen kann, ob bei einer Zuweisung eine echte Kopie notwendig ist oder nicht. Das entscheidende Kriterium zeigen die folgenden Zeilen:

```
ergstr = stra;        // Die rechte Seite ist ein L-Wert (siehe Seite 953).
ergstr = stra + strb; // Die rechte Seite ist ein R-Wert.
```

Um per Programm R-Werte explizit berücksichtigen zu können, wurde im neuen C++-Standard die Bindung temporärer Objekte an Referenzen durch die syntaktische Erweiterung `&&` für Referenzen auf R-Werte eingeführt:

```
void func(const Typ& t); // bindet an L-Wert
void func(Typ&& t);      // bindet an R-Wert
```

Der `const`-Qualifizierer fehlt typischerweise, weil Änderungen an den temporären Objekten möglich sein sollen (siehe unten). Beide Referenztypen werden über den normalen Mechanismus des Überladens von Funktionen eingebunden.

Um im Folgenden die Effekte am Beispiel zu zeigen, wird als Basis eine einfache String-Klasse `StringTyp`, die nur die wichtigsten Operationen enthält, vorausgesetzt. Diese Klasse wird anschließend um Referenzen auf R-Werte erweitert.

Listing 22.1: Klasse StringTyp

```
// cppbuch/k22/move/einfacherstring.h
#ifndef EINFACHERSTRING_H
#define EINFACHERSTRING_H
#include<cstddef>                // size_t
#include<cstring>                // strlen()
#include<iostream>               // ostream

class StringTyp {
public:
   StringTyp(const char* s)                // Konstruktor
      : len(strlen(s)), start(new char[len+1]) {
      strcpy(start, s);
   }

   StringTyp(const StringTyp& s)           // Kopierkonstruktor
      : len(s.len), start(new char[len+1]) {
      strcpy(start, s.start);
   }

   ~StringTyp() {                          // Destruktor
      delete [] start;
   }

   StringTyp& operator=(StringTyp s) {     // Zuweisungsoperator
      swap(s);
      return *this;
   }

   StringTyp& operator+=(const StringTyp& s) { // Verkettungssoperator
      char* temp = new char[len + s.len +1]; // neuen Platz beschaffen
      strcpy(temp, start);                 // Teil 1 kopieren
      strcpy(temp + len, s.start);         // Teil 2 kopieren
      delete [] start;                     // alten Platz freigeben
      len += s.len;            // Verwaltungsinformation aktualisieren
      start = temp;
      return *this;
   }

   const char* c_str() const { return start;} // C-String zurückgeben
private:
   size_t len;                             // Länge
   char *start;                            // Zeiger auf den Anfang
   void swap(StringTyp& m) {               // vertauschen
      std::swap(start, m.start);
      std::swap(len, m.len);
   }
};

// Ausgabeoperator
inline std::ostream& operator<<(std::ostream &os, const StringTyp& s) {
   os << s.c_str();
```

```
   return os;
}

// binärer Verkettungsoperator
inline StringTyp operator+(StringTyp a, const StringTyp& b) {
   return a += b;
}
#endif
```

Hinweise

Um das Beispiel auf den Effekt der Referenzen auf R-Werte zu beschränken, wurden optimierende Maßnahmen wie die Wiederbenutzung von Speicher oder Vorhalten von Speicher, wie in Abschnitt 6.1.1, beschrieben, weggelassen.

In der Datei *cppbuch/k22/move/einfacherstring.h* auf der DVD sind zusätzlich Testausgaben eingebaut, die die Aufrufe und die Anzahl der `new`-Operationen dokumentieren. Die Testausgaben sind durch auskommentieren der Zeile `#define TEST` abschaltbar.

Um die Wirkung mit oder ohne Referenzen auf R-Werte auf dem Computer nachvollziehen können, gibt es ein `main`-Programm mit einigen temporären Objekten:

Listing 22.2: main-Programm

```
// cppbuch/k22/move/main.cpp
#define EINFACH          // siehe unten
#ifdef EINFACH
#include"einfacherstring.h"
#else
#include"movingstring.h"
#endif
#include<iostream>
using namespace std;

int main() {
   StringTyp a("einA");
   StringTyp b("einB");
   StringTyp c("einC");
   cout << " Fall 1 : a = \"Hallo\" + b;" << endl;
   a = "Hallo" + b;

   cout << "\n Fall 2 :  a = b + \" hallo\";" << endl;
   a = b + " hallo";

   cout << "\n Fall 3 : a = StringTyp(\"guten\") + \" Tag\";" << endl;
   a = StringTyp("guten") + " Tag";

   cout << "\n Fall 4 : a = b + c;" << endl;
   a = b + c;

   cout << "\n Fall 5 : StringTyp neu = b + c;" << endl;
   StringTyp neu = b + c;
```

```
   cout << "\n Fall 6 :  neu = a + \"eins\" + \"zwei\" + \"drei\";" << endl;
   neu = a + "eins" + "zwei" + "drei";

   cout << "\na = " << a << endl;
   cout << "b = " << b << endl;
   cout << "c = " << c << endl;
   cout << "neu= " << neu << endl;

   cout << "\n Fall 7 : neu = a + b + c + neu;" << endl;
   neu = a + b + c + neu;
}
```

Performancekriterium

`new` und `delete`-Operationen sind zeitraubend. Im Folgenden wird die Anzahl eingesparten der `new`-Operationen als Maßstab für die Verbesserung der Performance genommen. Das `main`-Programm verursacht 50 `new`-Operationen, wenn *einfacherstring.h* inkludiert wird. Diese Zahl wird durch Referenzen auf R-Werte deutlich reduziert, wie unten zu sehen.

Um diese Referenzen im konkreten Fall der Klasse `StringTyp` nutzen zu können, werden unter anderem die Deklarationen

```
StringTyp(StringTyp&&);            // bewegender (nicht kopierender) Konstruktor
StringTyp& operator=(StringTyp&&); // bewegende Zuweisung
```

benötigt. Die Änderungen der Klasse `StringTyp` sind in der Datei *cppbuch/k22/move/movingstring.h* berücksichtigt. Im `main`-Programm oben wird mit dem Makro `EINFACH` entschieden, welche der beiden Dateien inkludiert wird. Um den Effekt der Referenzen auf R-Werte ungestört sehen zu können, wird die Wegoptimierung der Konstruktoraufrufe abgeschaltet. Das heißt, *main.cpp* wird mit der Option `-fno-elide-constructors` übersetzt, wenn `make` aufgerufen wird (siehe *cppbuch/k22/move/makefile*).

22.2.1 Bewegender Konstruktor

Der bewegende Konstruktor (englisch *moving constructor*) heißt so, weil er das Objekt anscheinend *bewegt*, nicht kopiert. Tatsächlich übernimmt er nur die Daten des Parameter-Objekts:

```
// Auszug aus cppbuch/k22/move/movingstring.h
StringTyp(StringTyp&& s)          // bewegender Konstruktor
   : len(s.len), start(s.start) { // Speicher übernehmen
   s.start = 0;                   // nicht vergessen! (siehe Text)
}
```

Der Konstruktor beschafft keinerlei Speicherplatz auf dem Heap und ist daher sehr schnell. Das Nullsetzen des Zeigers `s.start` ist unbedingt notwendig, damit der `delete []`-Aufruf des Destruktors von `s` unschädlich bleibt. Bei der Erzeugung des neuen Objekts werdem dem temporären Objekt, das danach nicht mehr gebraucht wird, die Ressourcen (d.h. der zugewiesene Speicher) »gestohlen« (englisch *resource stealing*).

Wenn der Parameter `s` ein L-Wert wäre, müsste er für eine weitere Verwendung erhalten bleiben, und man dürfte ihm nicht die Ressourcen entziehen. In diesem Fall ruft der Compiler den normalen Konstruktor `StringTyp(const StringTyp& s)` auf.

22.2.2 Bewegender Zuweisungsoperator

Der bewegende Zuweisungsoperator unterscheidet sich vom normalen Zuweisungsoperator dadurch, dass keine Kopie angelegt wird. Wie bei dem Kopierkonstruktor wird die Ressource des Parameters in Besitz genommen. Damit kein Speicherleck entsteht, muss der vorher belegte Speicher freigegeben werden.

```
// Auszug aus cppbuch/k22/move/movingstring.h
StringTyp& operator=(StringTyp&& s) { // bewegender Zuweisungsoperator
   delete [] start;                   // alten Speicher freigeben
   start = s.start;                   // Ressource aneignen
   s.start = 0;                       // wegen Destruktor von s
   len = s.len;
   return *this;
}
```

Die Variante mit `swap()` ist natürlich auch möglich. Die Funktion `swap()` vertauscht die Daten des Objekts `*this` mit denen von `s`. Damit wird erreicht, dass `*this` die gewünschten Daten erhält und der Destruktor von `s` die alten Daten von `*this`, die sich nunmehr in `s` befinden, korrekt löscht. Die Parameterübergabe per `&&` kann hier als zerstörender Lesevorgang aufgefasst werden. Da es sich um ein temporäres Objekt handelt, ist dies kein Problem.

```
// Variante
StringTyp& operator=(StringTyp&& s) { // bewegender Zuweisungsoperator
   swap(s);       // Ressourcen und Verwaltungsinformationen tauschen
   return *this;
}
```

Der bisher verwendete Zuweisungsoperator (`operator=(StringTyp)` von Seite 592) übergibt den Parameter per Wert, was für temporäre *und* andere Objekte möglich ist. Wenn der bewegende Zuweisungsoperator hinzukommt, kann der Compiler nicht entscheiden, welcher Zuweisungsoperator verwendet werden soll. Aus diesem Grund muss der Zuweisungsoperator `operator=(StringTyp)` geändert werden:

```
// Auszug aus cppbuch/k22/move/movingstring.h
// geändert! Jetzt Übergabe per const& statt per Wert
StringTyp& operator=(const StringTyp& s) {
   StringTyp tmp(s);
   swap(tmp);
   return *this;
}
```

Dieser Zuweisungsoperator wird nunmehr für L-Werte aufgerufen, der andere Zuweisungsoperator (`operator=(StringTyp&&)`) für R-Werte.

22.3 Ein effizienter binärer Plusoperator

Die im vorangehenden Abschnitt eingeführten Elemente »bewegender Konstruktor« und »bewegender Zuweisungsoperator« reduzieren die Zahl der `new`-Operationen von 50 auf 43. Damit ist noch nicht das Problem aller überflüssigen temporären Objekte gelöst. Es gibt vier Möglichkeiten der Verkettung zweier Strings, die L-Werte oder R-Werte sein können (`a`, `b` und `c` sind String-Objekte):

```
a = "Hallo" + b;                          // R-Wert L-Wert
a = b + " hallo";                         // L-Wert R-Wert
a = StringTyp("guten") + " Tag";          // R-Wert R-Wert
a = b + c;                                // L-Wert L-Wert
```

Hier sind Objekte mit Namen (L-Werte), aber auch temporäre Objekte beteiligt.

Der binäre `+`-Operator kann bei der Rückgabe mit einer Typumwandlung vom bewegenden Konstruktor profitieren:

```
StringTyp operator+(const StringTyp& x, const StringTyp& y) { // Verketten
   return static_cast<StringTyp&&>(StringTyp(x) += y);
}
```

Da die Wegoptimierung des Kopierkonstruktors vom C++Standard nicht vorgeschrieben wird, wird hier nachgeholfen, indem durch die Typumwandlung definitiv der bewegende Kopierkonstruktor aufgerufen wird. Achtung: Der Rückgabetyp ist `StringTyp`, nicht `StringTyp&&`! Der Unterschied: Im ersten Fall tritt das Objekt an die Stelle des Funktionsaufrufs, im zweiten Fall eine Referenz auf das (lokale!) Objekt, das nach Ende der Funktion dann nicht mehr existiert. Mit dieser Typumwandlung reduziert sich die Anzahl der `new`-Operationen für das obige `main`-Programm auf nur noch 32!

Es würde in allen vier Fällen der (normale) Kopierkonstruktor jeweils nur noch einmal aufgerufen, sowie jeweils einmal der Moving-Konstruktor und der Moving-Zuweisungsoperator.

std::move()

An dieser Stelle sei die Standardfunktion `move()` erwähnt, die dieselbe Typumwandlung bewirkt:

```
StringTyp operator+(const StringTyp& x, const StringTyp& y) { // Operator 1
   return std::move(StringTyp(x) += y);
}
```

Die Funktion `move(par)` kann dann vorteilhaft eingesetzt werden, wenn ihr Parameter `par` danach definitiv nicht mehr benötigt wird. Falls der Parameter ein L-Wert sein sollte, wäre eine weitere Verwendung kritisch – schließlich sind ihm möglicherweise die Ressourcen entzogen worden!

Mehr zu `move` lesen Sie in Abschnitt 27.2.

22.3.1 Kopien temporärer Objekte eliminieren

Der Operator `operator+(const StringTyp&, const StringTyp&)` berücksichtigt keine temporären Objekte in der Parameterliste. Wenn die Eigenschaft R-Wert der temporären Objekte ausgenutzt wird, ergeben sich drei weitere Varianten des +-Operators:

```
inline StringTyp operator+(StringTyp&& x, const StringTyp& y) { // Operator 2
   return std::move(x += y);
}

inline StringTyp operator+(const StringTyp& x, StringTyp&& y) { // Operator 3
   // return StringTyp(x) += y; besser, weil ein new weniger:
   return std::move(y.frontinsert(x));
}

inline StringTyp operator+(StringTyp&& x, StringTyp&& y) { // Operator 4
   return std::move(x += y);
}
```

Zu Operator 3: Der Aufruf `y.frontinsert(x)` fügt den String `x` *am Anfang* von `y` ein. Dafür ist nur eine `new`-Operation notwendig. `StringTyp(x) += y;` hingegen benötig ein `new` zur Konstruktion der temporären Kopie von `x` und ein weiteres für den `+=`-Operator. Die Funktion `frontinsert()` finden Sie in der Datei *cppbuch/k22/move/movingstring.h*.

Den Operatoren 2 bis 4 wird jeweils mindestens ein temporäres Objekt übergeben. Wenn mit diesem in der Operatorfunktion gearbeitet und die Referenz per `move()` darauf zurückgegeben wird, erübrigt sich der normale Kopierkonstruktor vollends. Es wird der bewegende Kopierkonstruktor aufgerufen. Es bleibt nur noch der bewegende Zuweisungsoperator – eine erhebliche Verbesserung! Sie finden das vollständige Programm im Verzeichnis *cppbuch/k22/move*.

Mit diesen Operatoren reduziert sich die Anzahl der `new`-Operation auf 24, also weniger als die Hälfte der anfänglichen 50!

22.3.2 Verbesserung durch verzögerte Auswertung

Nur im Fall des Operators 1 von oben ist einmal der Aufruf des Kopierkonstruktors notwendig, weil keins der beiden Argumente ein R-Wert ist. Das erste Argument durch eine Übergabe per Wert zu ersetzen, würde nichts bringen, weil es einen Compilations-Konflikt mit Operator 2 geben würde. Am besten wäre es, wenn dieses funktionslokale Objekt gar nicht erst erzeugt werden müsste und das Ergebnis direkt an den Zielort geschrieben werden könnte. Dann wäre keinerlei Kopierkonstruktoraufruf notwendig. In der Klasse `StringTyp` ist diese effiziente Variante etwa mit einer Member-Funktion realisierbar, in der die Parameter per Referenz übergeben werden:

```
void StringTyp::verketten(const StringTyp& x, const StringTyp& y) {
   char* temp = new char [x.len + y.len + 1]; // Platz beschaffen
   strcpy(temp, x.start);          // Teil 1 kopieren
   strcpy(temp + x.len, y.start); // Teil 2 kopieren
   delete [] start;               // alten Speicherplatz freigeben
   start = temp;                  // Verwaltungsdaten aktualisieren
   len = a.len + b.len;
}
```

Der Aufruf `ergstr = stra + strb;` würde einfach durch den Funktionsaufruf `ergstr.verketten(stra, strb);` ersetzt werden. Die Funktion vermeidet jeden Kopierkonstruktoraufruf und ist daher die schnellere Lösung. Aber – ist es denn wirklich notwendig, aus Performance-Gründen auf das C++-Markenzeichen »Überladen von Operatoren« zu verzichten? Die Antwort ist nein, wenn wir in die C++-Trickkiste greifen.

Ein binärer Operator weiß nicht, was mit seinem Ergebnis geschieht: Deswegen muss er ein temporäres Ergebnisobjekt zur Verfügung stellen, könnte man annehmen. Tatsächlich aber zeigt der Einsatz nur zwei typische Benutzungsweisen:

```
StringTyp neu = stra + strb; // Konstruktion eines neuen Objekts
ergstr = stra + strb;        // Zuweisung an ein vorhandenes Objekt
```

Der Aufruf des Kopierkonstruktors wäre vermeidbar, wenn es gelänge, die Beschaffung von Speicher und das Kopieren der Daten in den Kopierkonstruktor zu verlagern, die Auswertung der +-Operation also zu verzögern. Das kann mit dem Trick nach [Str], Kap. 22, erreicht werden, in `operator+()` nur die Referenzen der beteiligten Objekte zu übertragen und sonst nichts zu tun. Die eigentliche Arbeit muss dann im Kopierkonstruktor bzw. im Zuweisungsoperator erledigt werden. Die Übertragung wird mit einem Hilfstyp `Plus` vorgenommen, der nur ein Paar Referenzen auf die beteiligten Objekte in sich trägt. (Sie finden das vollständige Programm im Verzeichnis *cppbuch/k22/plus*.)

```
// Auszug aus cppbuch/k22/plus/movingstring.h
class StringTyp;
typedef std::pair<const StringTyp&, const StringTyp&> Plus;
```

`pair<U, V>` ist ein Standard-Template (Header `<utility>`, siehe Seite 750), das ein Wertepaar kapselt. Das Paar kann auch aus Elementen verschiedenen Typs bestehen. Für die Template-Parameter wird hier `StringTyp&` eingesetzt. Der Trick besteht darin, den Operator 1 oben so umzubauen, dass er den String nicht verkettet, sondern nur die Adressen der Parameter weiterleitet:

```
Plus operator+(const StringTyp& x, const StringTyp& y) { // Operator 1a
    return Plus(x, y);
}
```

Das Ergebnis der Verkettung führt entweder zu einem neuen Objekt, oder es wird einem anderen Objekt zugewiesen. Es werden also ein entsprechender Konstruktor und ein Zuweisungsopeartor benötigt. Beide erledigen die Verkettung, wobei der Zuweisungsoperator auf den Konstruktor zugreift. Nur im Konstruktor gibt es ein `new`:

```
StringTyp(const Plus& pa)  // Konstruktor für Plus
   : len(pa.first.len + pa.second.len),
     start(new char [len + 1]) {               // Platz beschaffen
   strcpy(start, pa.first.start);              // Teil 1 kopieren
   strcpy(start + pa.first.len, pa.second.start); // Teil 2 kopieren
}

StringTyp& operator=(const Plus& pa) { // Plus auswerten
   StringTyp temp(pa); // Konstruktor für Plus-Argument
   swap(temp);         // *this mit temp vertauschen
   return *this;
}
```

Damit wird im `main`-Beispiel die Anzahl der `new`-Operationen auf nunmehr 21 verringert. Auf der Basis dieser Lösung wird der Ablauf der Anweisung

```
ergstr = stra + strb + strc;
```

analysiert. In den folgenden Zeilen sehen Sie die einzelnen Schritte der Abarbeitung.

```
ergstr = operator+(stra, strb) + strc; // Operator 1a von oben
ergstr = Plus(stra, strb) + strc;      // Ergebnis
```

Weil der `+`-Operator ein `StringTyp`-Objekt erwartet, wird `Plus(stra, strb)` umgewandelt:

```
ergstr = StringTyp(Plus(stra, strb)) + strc;
```

Das Ergebnis der Umwandlung ist ein temporäres Objekt, hier `TEMP1` genannt. Dazu passend kommt der Operator 2 von oben (`operator+(StringTyp&&, const StringTyp&)`) zur Geltung:

```
ergstr = operator+(TEMP1, strc);
```

Dessen Ergebnis ist wieder ein temporäres Objekt, hier `TEMP2` genannt. `TEMP2` wird Parameter des bewegenden Zuweisungsoperators:

```
ergstr.operator=(TEMP2);
```

Das Ergebnis dieser Operation ist die veränderte Variable `ergstr`. Im Vergleich zu dem fiktiven Funktionsauf

```
ergstr.verketten(stra, strb);
```

von Seite 591 ist die Performance nur um den Aufruf des Konstruktors und Kopierkonstruktors der Klasse `Plus` schlechter. Da nur zwei Referenzen erzeugt bzw. kopiert werden, ist der zusätzliche Aufwand im Vergleich zur Speicherbeschaffung mit `new` und der Kopie der Strings unerheblich, und der gewünschte Performance-Gewinn ist erreicht. Der zusätzliche Konstruktor `StringTyp(const Plus&)` käme bei der Erzeugung eines `StringTyp`-Objekts aus zwei anderen zum Tragen, zum Beispiel

```
StringTyp verkettet = einString + zweiterString;
```

Ausblick

Für mehr als zwei rechte Operanden lässt sich eine Lösung gänzlich ohne temporäre Objekte finden, wenn Expression Templates [Ve95a] verwendet werden. Expression Templates dienen dazu, einen Ausdruck zur Compilationszeit mithilfe der Template-Metaprogrammierung, die in Abschnitt 6.4 (Seite 251) kurz beschrieben wird, zu optimieren. Eine andere Möglichkeit ist, anstelle von `Plus` ein Tupel (Klasse `tuple` von Seite 752) einzusetzen. Im Gegensatz zu `Plus` kann ein Tupel-Objekt eine beliebige Anzahl von Parametern speichern. Abschnitt 24.10.1 zeigt diesen Ansatz für eine 2-dimensionale Matrix.