Manipulierte Ströme

Ein- und Ausgabefunktionen in C++ selbst erweitern

Stefan Kuhlins

Lehrstuhl für Wirtschaftsinformatik III
Universität Mannheim
68131 Mannheim
stefan@kuhlins.de

17. Juli 1995

Zusammenfassung

Aller Anfang ist schwer, das gilt auch für den Umstieg von C auf C++. Insbesondere die neuen Ein- und Ausgaberoutinen erfordern einiges Umdenken. Hat man sie erst einmal durchschaut, so sind vielfältige Erweiterungsmöglichkeiten der Lohn.

Einleitung

Die von C her bekannten `stdio`-Funktionen gibt es in C++ immer noch. Wer auf C++ umsteigt, wird vielleicht versucht sein, seine Ein- und Ausgaben erst einmal über die gewohnten Funktionen abzuwickeln. Schließlich ist die neue Notation mittels IOStreams auf den ersten Blick unhandlicher, und alte `printf`-Hasen finden sie auch deutlich unübersichtlicher. Doch sollte objektorientierte Programmierung nicht bei der so wichtigen Ein- und Ausgabe enden, und so hat C++ in Form der IOStreams-Library einen der Sprache angemessenen Mechanismus hierfür mitbekommen.

An die Stelle der vordefinierten Streams `stdin`, `stdout` und `stderr` treten in C++ die IOStreams `cin`, `cout` und `cerr`. Das klassische 'Hello, world'-Programm sieht damit im Kern so aus:

```
cout << "Hello, world\n";
```

Die Rolle der Funktion `printf` übernimmt der Operator `<<` und analog dient der Operator `>>` der Eingabe von `cin` oder anderen IStreams. Möchte man mit `printf` andere Datentypen als Strings ausgeben, so funktioniert das mittels eines Formatierungs-Strings, den die Funktion als erstes Argument erwartet. Darin steht `%d` für einen Integer, `%f` für eine Fließkommazahl und so weiter.

```
#include <iostream.h>

class Bruch {
friend ostream& operator<<(ostream&, const Bruch&);
public:
  Bruch(int zz, int nn=1) : z(zz), n(nn) {}
  // ...
private:
  int z, n;
};

ostream& operator<<(ostream& os, const Bruch& b) {
  return os << b.z << '/' << b.n;
}

int main() {
  Bruch b(2, 3);
  cout << b << endl;
  return 0;
}
```

Abbildung 1: Benutzerdefinierter Ausgabeoperator

Allesfresser

Ganz anders bei C++: Ein und derselbe Ausgabeoperator schluckt alle Datentypen gleichermaßen. Er ist für die verschiedenen Typen überladen, das heißt, für jeden Typ gibt es eine eigene Definition des Operators `<<`. Der Compiler kann so automatisch für jeden Typ die geeignete Ausgabefunktion auswählen. Dadurch ist auf elegante Weise ein altes Problem von `printf` und Konsorten aus der Welt geschafft, nämlich die unzureichende Typprüfung. Wenn bei `printf` die tatsächlich angegebenen Parameter nicht genau zum Formatierungs-String passen, erhält man in der Regel statt einer Fehlermeldung beim Übersetzen einen Müllhaufen zur Laufzeit.

IOStreams haben noch einen weiteren Vorteil: Ohne den Quellcode der Library ändern zu müssen, kann der Benutzer sie für eigene Datentypen erweitern. Soll ein Programm beispielsweise mit Brüchen rechnen, um Rundungsfehler auszuschließen, so liegt die Definition einer Klasse nahe, die Zähler und Nenner separat speichert. Jetzt braucht man nur noch den Operator `<<` für Brüche zu überladen, und schon lassen sie sich genau wie alle anderen Datentypen ausgeben (siehe Abb. 1).

In ihrem Formatierungs-String versteht die Funktion `printf` allerlei Flags, die das Aussehen der Ausgabe beeinflussen. Beispielsweise dienen die Steuerzeichen `%u`, `%h` und `%o` dazu, vorzeichenlose `int`s dezimal, hexadezimal oder oktal auszugeben. Bei IOStreams merkt sich der Stream in einer Zustandsvariablen, mit welcher Zahlenbasis er arbeiten soll. Um eine Zahl hexadezimal zu Papier zu bringen, muß man daher den Ausgabe-Stream zunächst auf Hex umstellen und sie dann mit `<<` ausgeben:

```
cout.flags(ios::hex);
cout << 100;
```

Gegenüber der `printf`-Methode ist dies reichlich mühsam, und zum Glück gibt es eine einfachere Notation, die dasselbe bewirkt. Ein sogenannter Manipulator namens `hex` stellt, wenn man ihn ausgibt, sozusagen als Seiteneffekt den betroffenen Stream auf hexadezimale Darstellung um, ohne selbst eine Ausgabe zu erzeugen. Analog dazu sind die Manipulatoren `oct` für oktale und `dec` für dezimale Darstellung definiert. Mit ihrer Hilfe lassen sich nun Ausgaben in verschiedenen Zahlenbasen innerhalb einer Anweisung beliebig kombinieren, indem man sie einfach hintereinander schreibt:

```
cout << "0x" << hex << 100 << '=' << dec << 100;
```

Auch die übrigen Formatierungsmöglichkeiten von `printf` finden sich bei den IOStreams wieder. In der Header-Datei `iomanip.h` sind diverse Manipulatoren deklariert, die jeweils einen Parameter erwarten. Beispielsweise stellt `setprecision(x)` die Anzahl der auszugebenden Nachkommastellen ein, während `setw(x)` die Breite der Ausgabe bestimmt. Eine Handvoll weiterer Manipulatoren sind in [1] und [2] beschrieben.

Selbstgestrickt

Ein besonderer Reiz der IOStreams liegt darin, eigene Manipulatoren zu definieren. Die Abbildung 2 zeigt einen Manipulator namens `startline`, der eine fortlaufende Zeilennummer ausgibt. Er ist als Funktion definiert, die eine Referenz auf einen `ostream` als Argument und Rückgabewert besitzt. Um den neuen Manipulator mittels `<<` ausgeben zu können, sind keine weiteren Vorkehrungen erforderlich, da `iostream.h` bereits eine geeignet überladene Version des Operators enthält.

Was 'geeignet' in diesem Fall bedeutet, wird klar, wenn man sich vor Augen führt, wie die Anwendung des Manipulators genau erfolgt. Sein Name taucht ohne Klammern auf, steht also für die Adresse der Funktion. Vor die Aufgabe gestellt, eine Adresse auf eine Funktion genau dieses Typs auszugeben, ruft der Operator `<<` diese einfach auf und übergibt ihr 'seinen' Ausgabe-Stream als Parameter. Nun kann sie den betroffenen Stream nach Herzenslust manipulieren.

Während dies noch relativ einfach zu bewerkstelligen war, gestaltet sich die Definition eines Manipulators mit Argument wesentlich schwieriger. Beispielsweise wäre zur Ausgabe einer bestimmten Anzahl von Leerzeichen ein Manipulator `space` ganz nett, der als Argument die gewünschte Anzahl an Leerzeichen entgegennimmt: Um ihn analog zu `setw` verwenden zu können, sind einige Klimmzüge bei der Definition erforderlich. Wäre `space` eine Funktion wie `startline`, so würde die Anweisung `cout << space(5)` zunächst `space(5)` aufrufen, bevor der Operator `<<` zum Zuge käme. In diesem Fall könnte die Funktion aber nicht wissen, auf welchen Stream sie eigentlich ausgeben soll.

Man muß daher zu einem Trick greifen und `space` so definieren, daß es die gewünschte Ausgabe nicht selbst vornimmt, sondern irgendein Objekt liefert, das der Operator `<<` dann geeignet interpretiert. Beispielsweise kann `space`

```
#include <iostream.h>
#include <iomanip.h>

ostream& startline(ostream& os) {
  static int zeile = 1;
  return os << setw(3) << zeile++ << ": ";
}

int main() {
  cout << startline << "Beginn" << endl;
  for (int i=0; i<10; i++)
    cout << startline << i << endl;
  cout << startline << "Ende" << endl;
  return 0;
}
```

Abbildung 2: Ein Manipulator ohne Argument läßt sich ganz leicht als Funktion realisieren.

selbst eine Klasse sein. Diese erhält einen Konstruktor, der ein Argument vom Typ `int` erwartet. `space(5)` konstruiert dann ein temporäres Objekt der Klasse `space`, das sich in einem `private` Datenelement die Anzahl der auszugebenden Leerstellen merkt. Damit der Ausgabeoperator dieses versteht, muß man ihn schließlich für die Klasse `space` überladen (siehe Abb. 3).

Recycling

Sollen in einem Programm mehrere Manipulatoren mit Argumenten vorkommen, so liefert die oben geschilderte Methode für jeden eine eigene Klasse, wobei sich die einzelnen Klassen eigentlich nur in der Funktionalität des jeweiligen Ausgabeoperators unterscheiden. Ein Ziel der objektorientierten Programmierung ist es aber, das Rad nicht jedesmal neu zu erfinden, sondern möglichst viel bestehenden Code wiederzuverwenden.

Der nächste Schritt auf dem Weg dorthin ist es daher, die eigentliche Ausgabefunktion aus dem Ausgabeoperator auszulagern und sie indirekt über einen Zeiger aufzurufen. Ein Manipulator besteht dann aus drei Komponenten: einer Ausgabefunktion, die die eigentliche Arbeit erledigt, der Manipulatorfunktion selbst und einer Klasse, die ich `OMI` (für `OstreamManipulatorlnt`) genannt habe. Diese enthält, wie der Name schon andeuten soll, alles, was Ausgabemanipulatoren gemeinsam haben, jedenfalls solche mit einem `int` als Parameter. Für weitere Manipulatoren braucht man dann nur noch die Ausgabefunktion und die eigentliche Manipulatorfunktion zu schreiben und kann die Klasse `OMI` wiederverwenden.

Mit `OMI` funktioniert das `space`-Beispiel nun wie folgt: Die Funktion `space` liefert ein Objekt der Klasse `OMI`, das sie mit der gewünschten Anzahl von Leerzeichen und einem Zeiger auf die Ausgabefunktion, in diesem Beispiel `spaces`, initialisiert. Der Operator `<<` der nur einmal für die Klasse `OMI` überladen ist,

```
#include <iostream.h>

class space {
friend ostream& operator<<(ostream&, const space&);
public:
  space(int nn) : n(nn) {}
private:
  const int n;
};

ostream& operator<<(ostream& os, const space& s) {
  for (int i=0; i<s.n; i++)
    os << ' ';
  return os;
}

int main() {
  cout << "0123456\n";
  cout << '#' << space(5) << '#' << endl;
  return 0;
}
```

Abbildung 3: Damit ein Manipulator ein Argument versteht, realisiert man ihn im einfachsten Fall als Klasse.

kann dieses Objekt dann ausgeben, indem er über den darin gespeicherten Zeiger die richtige Funktion mit dem geeigneten Argument aufruft (siehe Abb. 4).

Schließlich läßt sich die Abstraktion noch einen Schritt weiter treiben, denn alle gleichartigen Manipulatoren haben gemeinsam, daß man für sie eine Funktion definieren muß, die die zugehörige Hilfsfunktion aufruft. Diese Gemeinsamkeit läßt sich in einer Applikatorklasse festhalten, die ich im Beispiel `AppI` genannt habe, und Manipulatoren sind dann als Objekte dieser Klasse realisiert.

Der Weg zu einem neuen Manipulator ist hiermit zwar gedanklich noch einen Schritt weiter, aber dafür erfordert jeder weitere Manipulator nur noch minimalen Aufwand. Man braucht lediglich die Hilfsfunktion zu definieren, die die eigentliche Arbeit erledigt, und den Manipulator dann als Objekt der Klasse `AppI` zu deklarieren. Der Konstruktor von `AppI` erwartet als Parameter die Adresse der Hilfsfunktion und speichert sie in einem privaten Datenelement. Wohlgemerkt, dies passiert nur einmal bei der Definition des Manipulators. Bei seiner Anwendung kommt der überladene Funktionsaufrufoperator `()` zum Zuge: er liefert wie gehabt das temporäre Objekt der Klasse `OMI`, das der Operator `<<` dann ausgibt (siehe Abb. 5).

Die Krönung

Der nächste und letzte verallgemeinerungswürdige Aspekt bezieht sich auf den Parameter der Manipulatoren. Alle bisher vorgestellten Versionen hatten einen Parameter vom Typ `int`; für andere Typen müßte man neue Klassen definie-

```
#include <iostream.h>

// --- gemeinsamer Teil fuer alle Manipulatoren ---
class OMI {
public:
  typedef int Arg;
  typedef ostream& (*Fkt)(ostream&, Arg);
  OMI(Fkt ff, Arg aa) : f(ff), a(aa) {}
  friend ostream& operator<<(ostream& os, const OMI& omi) {
    return omi.f(os, omi.a);
  }
private:
  const Fkt f;
  const Arg a;
};

// --- Definition des space-Manipulators ---
ostream& spaces(ostream& os, int n) {
  for (int i=0; i<n; i++)
    os << ' ';
  return os;
}

inline OMI space(int n) { return OMI(spaces, n); }

int main() {
  cout << "0123456\n";
  cout << '#' << space(5) << '#' << endl;
  return 0;
}
```

Abbildung 4: Für mehrere gleichartige Manipulatoren lohnt es sich, eine allgemeine Klasse mit einem Funktionszeiger zu definieren.

ren. Genau für diesen Fall besitzt C++ inzwischen Templates. Dabei handelt es sich sozusagen um eine verallgemeinerte Klassendefinition, bei der eine einzige Schablone eine ganze Menge gleichartiger Klassen definiert, die sich nur durch die verwendeten Typen unterscheiden. Aus den Klassen `OMI` und `AppI`, die nur für Argumente des Typs `int` gelten, werden die Template-Klassen `OM` und `App`, die für beliebige Typen gut sind (siehe Abb. 6). Die vorerst letzte Version des Beispielprogramms funktioniert damit genau so, wie die vordefinierten Manipulatoren in einer modernen IOStreams-Library implementiert sind. Demjenigen, der sich Gedanken über die Performance solcher Konstrukte macht, sei gesagt, daß bis auf die Hilfsfunktion `spaces` alle Funktionen `inline` definiert sind und auch umgesetzt werden können, so daß im Endeffekt kompakter Code herauskommt.

Einen Großteil der Programmierarbeit kann man sich schließlich ersparen, indem man den Code aus `iomanip.h` wiederverwendet. Ein Blick in diese Datei verrät, ob der Bibliotheksentwickler bereits mit den fortschrittlichen Templates

oder noch per `#define` mit Makros gearbeitet hat. Je nach Compiler heißt die Applikatorklasse `oapp<tvp>` oder `OAPP(typ)`, die Klasse für das temporäre Manipulatorobjekt `omanip<typ>` beziehungsweise `OMANIP(typ)`. Damit reduziert sich ein neuer Manipulator dann wirklich auf wenige Zeilen, ohne daß man die dahinterstehende Komplexität im Programm sieht (siehe Abb. 7).

Lediglich für Manipulatoren mit mehreren Argumenten muß man selbst Hand anlegen und analog zu den obigen Beispielen eigene Klassen definieren. Solange nur ein solcher Manipulator vorkommt, lohnt der Aufwand der Abstraktion mit Applikatorklasse und Templates nicht, und die eingangs gezeigte einfache Implementation des Manipulators als Klasse ist am übersichtlichsten (siehe Abb. 8). Erst bei mehreren Manipulatoren ähnlicher Bauart zahlen sich die komplizierteren der vorgestellten Möglichkeiten aus.

Literatur

[1] Martin Schader und Stefan Kuhlins, Programmieren in C++, 3. Auflage, Springer-Verlag, 1995

[2] Steve Teale, C++ IOStreams Handbook, Addison-Wesley, 1993

```
#include <iostream.h>

// --- gemeinsamer Teil fuer alle Manipulatoren ---
class OMI {
public:
  typedef int Arg;
  typedef ostream& (*Fkt)(ostream&, Arg);
  OMI(Fkt ff, Arg aa) : f(ff), a(aa) {}
  friend ostream& operator<<(ostream& os, const OMI& omi) {
    return omi.f(os, omi.a);
  }
private:
  const Fkt f;
  const Arg a;
};

class AppI {
public:
  AppI(OMI::Fkt ff) : f(ff) {}
  OMI operator()(OMI::Arg n) const { return OMI(f, n); }
private:
  const OMI::Fkt f;
};

// --- Definition des space-Manipulators ---
ostream& spaces(ostream& os, int n) {
  for (int i=0; i<n; i++)
    os << ' ';
  return os;
}

const AppI space(spaces);

int main() {
  cout << "0123456\n";
  cout << '#' << space(5) << '#' << endl;
  return 0;
}
```

Abbildung 5: Durch eine zusätzliche Applikatorklasse wird die Definition weiterer Manipulatoren noch einfacher.

```
#include <iostream.h>

// --- gemeinsamer Teil fuer alle Manipulatoren ---
template<class Arg>
class OM {
public:
  typedef ostream& (*Fkt)(ostream&, Arg);
  OM(Fkt ff, Arg aa) : f(ff), a(aa) {}
  friend ostream& operator<<(ostream& os, const OM<Arg>& om) {
    return om.f(os, om.a);
  }
private:
  const Fkt f;
  const Arg a;
};

template<class Arg>
class App {
public:
  App(OM<Arg>::Fkt ff) : f(ff) {}
  OM<Arg> operator()(Arg n) const { return OM<Arg>(f, n); }
private:
  const OM<Arg>::Fkt f;
};

// --- Definition des space-Manipulators ---
ostream& spaces(ostream& os, int n) {
  for (int i=0; i<n; i++)
    os << ' ';
  return os;
}

const App<int> space(spaces);

int main() {
  cout << "0123456\n";
  cout << '#' << space(5) << '#' << endl;
  return 0;
}
```

Abbildung 6: Mit Hilfe von Templates lassen sich die Manipulatorklassen für beliebige Datentypen verallgemeinern.

```
#include <iostream.h>
#include <iomanip.h>

ostream& spaces(ostream& os, int n) {
  for (int i=0; i<n; i++)
    os << ' ';
  return os;
}

#ifdef OAPP
OAPP(int) space(spaces);
#else
oapp<int> space(spaces);
#endif

int main() {
  cout << "0123456\n";
  cout << '#' << space(5) << '#' << endl;
  return 0;
}
```

Abbildung 7: So einfach kann die Erstellung eines eigenen Manipulators sein, wenn man die vordefinierten Klassen der IOStreams-Library verwendet.

```
#include <iostream.h>

class replicate {
friend ostream& operator<<(ostream&, const replicate&);
public:
  replicate(char cc, int nn) : c(cc), n(nn) {}
private:
  const char c;
  const int n;
};

ostream& operator<<(ostream& os, const replicate& r) {
  for (int i=0; i<r.n; i++)
    os << r.c;
  return os;
}

int main() {
  cout << "0123456\n";
  cout << '#' << replicate('-', 5) << '#' << endl;
  return 0;
}
```

Abbildung 8: Manipulator mit mehreren Argumenten als Klasse