10

Dateien und Ströme

Dieses Kapitel behandelt die folgenden Themen:

- Formatierung der Ausgabe
- Funktionsweise der Standardeingabe
- Manipulatoren – Wirkungsweise und Konstruktion
- Fehlerbehandlung bei der Ein- und Ausgabe
- Status einer Datei
- Ansteuern bestimmter Positionen einer Datei
- Nutzung von cin für eigene Klassen
- Ausgabe in einen String umleiten

In Kapitel 2 wird die Ein- und Ausgabe einführend beschrieben. In diesem Kapitel wird das Thema vertieft, indem weitere nützliche Funktionen und Operatoren angegeben werden sowie die Formatierung von Daten besprochen und auf die Fehlerbehandlung eingegangen wird. Die C++-Standardbibliothek stellt die notwendigen Mechanismen für die Ein- und Ausgabe von Grunddatentypen zur Verfügung, erlaubt aber auch die Konstruktion eigener Ein- und Ausgabefunktionen für selbst definierte Klassen. Auf der untersten Ebene wird ein *stream* als Strom oder Folge von Bytes aufgefasst. Das Byte ist die Dateneinheit des Stroms, andere Datentypen wie `int`, `char*` oder `vector` erhalten erst durch die Bündelung und Interpretation von Bytesequenzen auf höherer Ebene ihre Bedeutung. Die Basisklasse heißt `ios_base`; aus ihr werden die anderen Klassen abgeleitet, wie Abbildung 10.1 zeigt. Die dort mit dem Wort `basic` beginnenden Klassen sind Templates, die für beliebige Zeichentypen geeignet sind, zum Beispiel Unicode-Zeichen oder andere »Wide Characters«, die auf Seite 51 erwähnt werden. Für den am häufigsten benötigten Datentyp `char` sind die Klassen durch Typdefinitionen wie

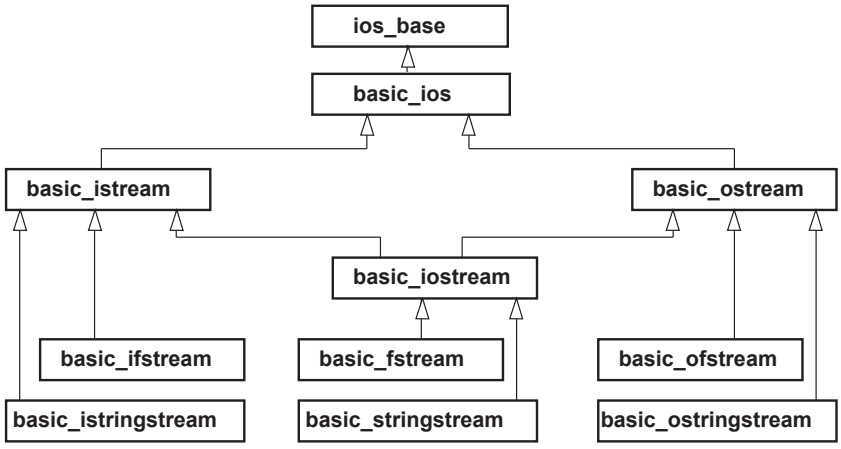

Abbildung 10.1: Hierarchie der Klassen-Templates für die Ein- und Ausgabe (Auszug)

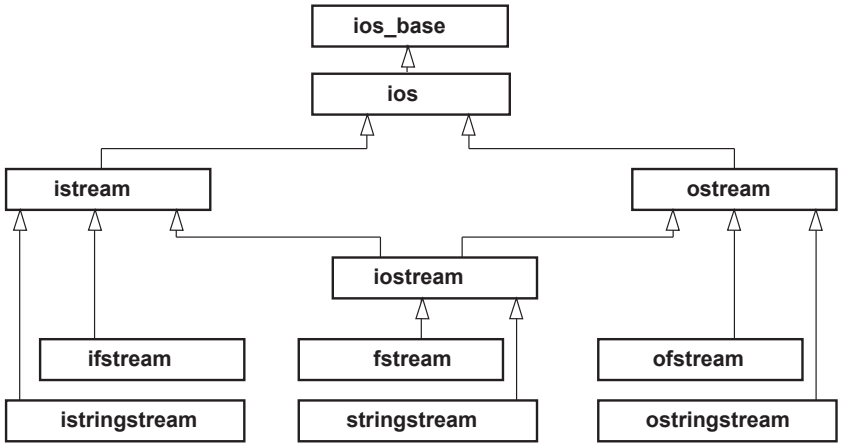

Abbildung 10.2: Spezialisierte Klassen für `char`

```
typedef basic_ofstream<char> ofstream;
```

spezialisiert worden, wie Abbildung 10.2 zeigt. Im Folgenden liegt der Schwerpunkt auf der Benutzung der Klassen. Die im Header `<iostream>` deklarierten Standardstreams sind Objekte dieser Klassen:

```
namespace std {                 // Auszug aus <iostream>
    extern istream cin;         // Standardeingabe
    extern ostream cout;        // Standardausgabe
    extern ostream cerr;        // Standardfehlerausgabe
    extern ostream clog;        // gepufferte Standardfehlerausgabe
}
```

10.1 Ausgabe

Die Klasse `ostream` umfasst überladene Operatoren für eingebaute Datentypen. Der Operator << dient dazu, ein Objekt eines internen Datentyps in eine Folge von ASCII-Zeichen zu verwandeln.

```
ostream& operator<<(const char*); // C-Strings
ostream& operator<<(char);
ostream& operator<<(int);
ostream& operator<<(float);
ostream& operator<<(double);
//... usw.
```

Der Rückgabetyp des Operators ist eine *Referenz* auf `ostream`. Das Ergebnis des Operators ist das `ostream`-Objekt selbst, sodass ein weiterer Operator darauf angewendet werden kann. Damit ist die Hintereinanderschaltung von Ausgabeoperatoren möglich:

```
cerr << "x= " << x;
```

wird interpretiert als

```
(cerr.operator<<("x= ")).operator<<(x);
```

Ausgabe benutzerdefinierter Typen

Überladen von Operatoren ermöglicht die Ausgabe beliebiger Objekte benutzerdefinierter Klassen. Ein ausführliches Beispiel dazu wird auf den Seiten 322 ff. gezeigt, wo es darum geht, Objekte der Klasse `rational` auszugeben. Nach diesem Muster sind eigenen Kreationen des <<-Operators keine Grenzen gesetzt.

Ausgabefunktionen

In der Klasse `ostream` sind weitere Elementfunktionen für `ostream`-Objekte definiert:

```
ostream& put(char);
```

gibt ein Zeichen aus. Wir haben `put()` bereits in Abschnitt 2.2 kennengelernt, wo die Funktion in einem Beispielprogramm zum Kopieren von Dateien eingesetzt wird. Die Funktion

```
ostream& write(const char*, size_t);
```

wird in Abschnitt 5.8 zur binären Ausgabe verwendet. Es gibt eine Menge zusätzlicher Funktionen, zum Beispiel zum Positionieren des Ausgabestroms auf eine bestimmte Stelle, zum Herausfinden der aktuellen Stelle, und weitere, auf die hier zum Teil eingegangen werden soll. Die Deklarationen dieser Funktionen finden Sie im Header `<iostream>`. Eine ausführliche Beschreibung findet sich in dem englischsprachigen Buch [KL]).

10.1.1 Formatierung der Ausgabe

Oft ist es notwendig, die Ausgabe besonders aufzubereiten, sei es als Tabelle, in der alle Spalten die gleiche Breite haben müssen, oder sei es, dass spezielle Zahlenformate

verlangt werden. Dieser Abschnitt stellt die wichtigsten Möglichkeiten zur Formatierung der Ausgabe vor.

Weite und Füllzeichen

Die Methode `width()` bestimmt die Weite der *unmittelbar folgenden* Zahlen- oder C-Stringausgabe. Dabei entstehende Leerplätze werden mit Leerzeichen aufgefüllt. Anstelle der Leerzeichen können mittels der Funktion `fill()` andere Füllzeichen definiert werden. Das Programmstück

```
cout.width(6);
cout.fill('0');
cout << 12 << '(' << 34 << ')';
```

erzeugt die Ausgabe 000012(34). Die Weite wurde auf 6 gesetzt, mit 0 als Füllzeichen. Weil die Weite aber bei jeder Ausgabe auf 0 zurückgesetzt wird, erscheint *(34)* ohne Füllzeichen. Es ist erkennbar, dass mindestens die notwendige Weite genommen wird, dass also eine zu kleine Angabe für die Weite die Ausgabe nicht beschränkt:

```
cout.width(4);
cout << 123456;
```

ergibt 123456, und nicht etwa 1234 oder ****.

Steuerung der Ausgabe über Flags

Ein *Flag* (englisch für *Flagge, Fahne*) ist ein Zeichen für ein Merkmal, das entweder vorhanden (Flag ist gesetzt) oder nicht vorhanden ist (Flag ist nicht gesetzt). Zur Formatsteuerung sind Flags des (compilerabhängigen) Datentyps `ios_base::fmtflags` definiert, der als Bitmaske benutzt wird. Die Tabelle 10.1 zeigt eine Aufstellung.

Weil die Klasse `ios` von `ios_base` erbt und im Folgenden nur die auf den Typ `char` spezialisierten Klassen der Ein- und Ausgabe betrachtet werden, wie die Abbildung 10.2 zeigt, können alle öffentlichen Attribute und Methoden auf die Klasse `ios` bezogen werden. `ios::fmtflags` ist dasselbe wie `ios_base::fmtflags`, weil `ios_base` kein Template ist. Statt `ios_base` wird im Folgenden nur noch `ios` geschrieben. Dieses Vorgehen hat den Vorteil der weitgehenden Kompatibilität mit früheren C++-Versionen und der einfacheren Schreibweise.

Im folgenden Programmbeispiel wird gezeigt, wie die Flags gelesen und gesetzt werden können. Die typische Voreinstellung mit Zweierpotenzen im Typ `fmtflags` definiert eine Bitleiste, deren einzelne Bits durch Oder-Operationen gesetzt werden. Die Funktion `flags()` dient zum Lesen und Setzen *aller* Flags, und `setf()` wird zum Setzen *eines* Flags verwendet.

```
ostream Ausgabe;
ios::fmtflags altesFormat;
ios::fmtflags neuesFormat = ios::left|ios::oct|ios::showpoint|ios::fixed;
// gleichzeitiges Lesen und Setzen aller Flags
altesFormat = Ausgabe.flags(neuesFormat);
// Setzen eines Flags
Ausgabe.setf(ios::hex);      // ungünstig, siehe unten
// gleichwertig damit ist:
```

Tabelle 10.1: Formateinstellungen für die Ausgabe

Name	Bedeutung
`boolalpha`	`true`/`false` alphabetisch ausgeben oder lesen
`skipws`	Zwischenraumzeichen ignorieren
`left`	linksbündige Ausgabe
`right`	rechtsbündige Ausgabe
`internal`	zwischen Vorzeichen und Wert auffüllen
`dec`	dezimal
`oct`	oktal
`hex`	hexadezimal
`showbase`	Basis anzeigen
`showpoint`	nachfolgende Nullen ausgeben
`uppercase`	E,X statt e,x
`showpos`	+ bei positiven Zahlen anzeigen
`scientific`	Exponential-Format
`fixed`	Gleitkomma-Format
	Puffer leeren (flush):
`unitbuf`	– nach jeder Ausgabeoperation
`stdio`	– nach jedem Textzeichen

```
Ausgabe.flags(Ausgabe.flags() | ios::hex);
// Zurücksetzen eines Flags
Ausgabe.unsetf(ios::hex);
```

Wenn ein Flag gesetzt werden soll, ist es besser, sicherheitshalber möglicherweise kollidierende Flags zurückzusetzen. Die Funktion `setf()` mit *zwei* Parametern sorgt dafür:

```
Ausgabe.setf(dieFlags, Maske);
```

bewirkt, dass zuerst alle Bits der Maske zurückgesetzt und danach die Flags gesetzt werden. Es gibt drei vordefinierte Masken für diesen Zweck, die in der Tabelle 10.2 zusammengefasst sind. Wenn die hexadezimale Ausgabe eingestellt werden soll, ist es daher besser, `Ausgabe.setf(ios::hex, ios::basefield)` zu schreiben, um gleichzeitig die möglicherweise gesetzten Flags `oct` oder `dec` zurückzusetzen.

Tabelle 10.2: Vordefinierte Bitmasken

Name	Wert
`adjustfield`	`left | right | internal`
`basefield`	`oct | dec | hex`
`floatfield`	`fixed | scientific`

Wenn in Ihrem System gelegentlich die Ausgabe nicht zum erwarteten Zeitpunkt geschieht, liegt es wahrscheinlich daran, dass `unitbuf` nicht voreingestellt ist. Manchmal ist es sinnvoll, den Benutzer eines Programms auf eine Wartezeit vorzubereiten:

```
// unitbuf zu Demonstrationszwecken zurücksetzen
cout.unsetf(ios::unitbuf);
cout << "langwierige Berechnung, bitte warten..\n";
// ... hier folgt die Berechnung
cout << "Ende der Berechnung" << endl;
```

Falls `unitbuf` nicht gesetzt ist, wie im Beispiel erzwungen, schlummert die Aufforderung zum Warten im Ausgabepuffer und erscheint erst *nach dem Ende* der Berechnung. `endl` bewirkt das Anhängen von `\n` sowie ein anschließendes Leeren des Ausgabepuffers mit `cout.flush()`. Das gewünschte Verhalten einer nicht gepufferten Ausgabe wird mit `cout.setf(ios::unitbuf)` erreicht.

Weite von Fließkommazahlen

Die Weite von Fließkommazahlen wird mit der Funktion `precision()` gesteuert, die die Anzahl der Ziffern bei Fließkomma-Ausgabe festlegt, sofern `fixed` oder `scientific` nicht gesetzt sind. Andernfalls legt `precision` die Anzahl der Nachkommastellen fest. Die eingestellte Anzahl ist gültig bis zum nächsten `precision()`-Aufruf. Mit dieser Funktion kann auch die aktuell eingestellte Ziffernzahl festgestellt werden. Das Programm

```
int vorherigeAnzahlDerZiffern = cout.precision();
cout.precision(8);
cout << 1234.56789 << " ";
cout << 1234.56789 << " ";   // precision bleibt erhalten
cout.precision(4);
cout << 1234.56789 << endl;
cout.precision(vorherigeAnzahlDerZiffern); // Wiederherstellung
```

erzeugt die Ausgaben (mit automatischer Rundung)

1234.5679 1234.5679 1235

Falls die Anzahl der Ziffern vor dem Komma den Wert von `precision` überschreitet, wird auf die wissenschaftliche Notation, also Ausgabe mit Exponent, umgeschaltet.

Anzahl der Nachkommastellen festlegen

Falls `fixed` oder `scientific` gesetzt ist, legt `precision()` die Anzahl der Nachkommastellen fest. Das Beispiel spricht für sich:

```
double f = 1234.123456789012345;
cout.setf(ios::scientific, ios::floatfield);
cout.precision(4);
cout << f << endl;          // 1.2341e+03
cout.setf(ios::fixed, ios::floatfield);
cout.precision(8);
cout << f << endl;          // 1234.12345679
```

10.2 Eingabe

Die Klasse `istream` enthält Operatoren für die in C++ eingebauten Grunddatentypen, die für eine Umwandlung der eingelesenen Zeichen in den richtigen Datentyp sorgen:

```
istream& operator>>(char*);      // C-Strings
istream& operator>>(char&);
```

```
istream& operator>>(float&);
istream& operator>>(int&);
//... usw.
```

Der Operator `>>` ist als überladener Operator definiert, wie wir ihn im Prinzip schon kennengelernt haben. Für einen Datentyp `T` (`T` steht für einen der Grunddatentypen) sei hier das Schema der Definition gezeigt:

```
istream& istream::operator>>(T& var) {
    // überspringe Zwischenraumzeichen
    // lies die Variable var des Typs T aus dem istream ein
    return *this;
}
```

Wie bei der Ausgabe bewirkt die Rückgabe einer Referenz auf `istream`, dass mehrere Eingaben verkettet werden können:

```
cin >> a >> b;          // ist gleichbedeutend mit
(cin >> a) >> b;        // und wird interpretiert als
(cin.operator>>(a)).operator>>(b);
```

Falls Zeichen oder Bytes eingelesen und Zwischenraumzeichen *nicht* ignoriert werden sollen, kann die Elementfunktion `istream& get()` genommen werden, die in mehreren überladenen Versionen bereitsteht, von denen eine ein einzelnes Zeichen einliest, wie das Beispiel zeigt. Zur Auswertung der `while`-Bedingung siehe Abschnitt 10.5.

```
// zeichenweises Kopieren der Standardeingabe (vgl. Beispiel Seite 97)
char c;
while(cin.get(c)) {
    cout << c;
}
// ebenfalls möglich ist:
while(cin.get(c)) {
    cout.put(c);
}
```

Zum sicheren Einlesen einer Zeichenkette in einen Pufferbereich wird einer anderen überladenen Version von `get()` der Zeiger auf den Pufferbereich und die Puffergröße mitgegeben, die wegen des abschließenden '`\0`'-Zeichens um eins größer als die Anzahl der maximal einzulesenden Elemente sein muss. Die Deklaration im Header `<iostream>` lautet `istream& get( char*, unsigned int, char t='\n');`. Die Zeichenkette wird bis zu einem festzulegenden Zeichen `t` übernommen (`t` steht für »Terminator«), wobei das Terminatorzeichen im Eingabestrom verbleibt. Es ist mit der Zeilenendekennung vorbesetzt. Beispiel:

```
// Zeile einlesen (max. N-1 Zeichen):
const unsigned int N = 100;
char buf[N];
cin >> buf;       // unsicher und Abbruch beim ersten Zwischenraumzeichen
cin.get(buf, N); // sicher wegen Längenbegrenzung auf N-1 Zeichen
// ... hier ggf. Terminatorzeichen lesen
```

Wegen der Vorbesetzung des Terminatorzeichens ist der Aufruf gleichbedeutend mit `cin.get(buf, N, '\n');`. Als letztes Zeichen trägt `get()` in `buf[]` die Stringendekennung

'\0' ein. Darüber hinaus gibt es einige weitere istream-Funktionen, von denen ein kleiner Teil hier aufgeführt ist:

```
istream& getline(char*, unsigned int, char t='\n');
```

Diese Funktion wirkt wie get(), aber das Terminatorzeichen wird gelesen (jedoch nicht mit in den Puffer übernommen).

```
istream& ignore(size_t n = 1, int t = EOF);
```

Diese Funktion liest und verwirft alle Zeichen des Eingabestroms, bis entweder das Terminatorzeichen oder n andere Zeichen gelesen worden sind. Das Terminatorzeichen verbleibt nicht im istream. EOF ist ein vordefiniertes Makro, das das Dateiende (englisch *end of file*) kennzeichnet und das häufig durch den Wert –1 repräsentiert wird. Ein Aufruf könnte zum Beispiel cin.ignore(max_Zeilenlaenge, '\n'); lauten.

```
istream& putback(char c);
```

Diese Funktion gibt ein Zeichen c an den istream zurück. Die Funktion

```
int get();
```

holt das nächste Zeichen und gibt es als int-Wert entsprechend seiner Position in der ASCII-Tabelle zurück. Bei EOF wird –1 zurückgegeben. Man könnte die Standardeingabe daher auf folgende Art kopieren:

```
int i;
while(cin.good() && (i = cin.get()) != EOF) { // d.h. weder EOF noch Lesefehler
   cout.put(static_cast<char>(i));
}
```

Eine andere Möglichkeit wäre die Abfrage mit eof(), eine Funktion, die true zurückgibt, wenn ein Leseversuch wegen Erreichens des Dateiendes erfolglos bleibt:

```
char c;
while(!cin.eof()) {  // besser: while(cin.good()) ...
   cin.get(c);
   if(!cin.fail()) { // EOF ist möglich, deswegen nicht if(cin.good())
      cout.put(c);
   }
}
```

cin.good() ist vorzuziehen, weil nicht nur EOF, sondern auch Lesefehler berücksichtigt werden. Eine Vorschau auf das nächste Zeichen im Eingabestrom wird durch die Funktion

```
int peek();
```

erlaubt. Die Wirkung von c = cin.peek() ist wie die Hintereinanderschaltung der Anweisungen (mit impliziter Typumwandlung)

```
c = cin.get(); cin.putback(c);
```

10.3 Manipulatoren

Manipulatoren sind Operationen, die direkt in die Ausgabe oder Eingabe zur Erledigung bestimmter Funktionen, zum Beispiel zur Formatierung, eingefügt werden. In diesem Abschnitt werden Manipulatoren nur für die Ausgabe beschrieben, das Prinzip lässt sich jedoch gleichermaßen auch für die Eingabe anwenden. Um eine `int`-Zahl oktal auszugeben, schreibt man

```
cout << oct << zahl << endl;
```

Die Wirkung ist genauso, als ob ein Flag zur Formatänderung gesetzt worden wäre:

```
cout.setf(ios::oct, ios::basefield);
cout << zahl << endl;
```

Sowohl `oct` als auch `endl` sind Manipulatoren. `endl` haben wir bereits kennengelernt. Wie funktioniert so ein Manipulator? Es gibt spezielle überladene Formen des Ausgabeoperators und verschiedene Funktionen, zum Beispiel `oct()` und `endl()`:

```
// Funktionen zur Klasse ostream
ostream& operator<<(ostream& (*fp)(ostream&));
ostream& operator<<(ios& (*fp)(ios&));
//.....
ostream& endl(ostream&);
// Funktion zur Klasse ios
ios& oct(ios&);
```

Der überladene Operator erwartet ein Argument vom Typ »Zeiger auf eine Funktion, die eine Referenz auf einen `ostream` als Parameter hat und eine Referenz auf einen `ostream` als Ergebnis zurückliefert«. Die Funktion `endl()` erfüllt genau dieses Kriterium, und die Funktion `oct()` entsprechend für die Klasse `ios`. Es werden in der einfachen Schreibweise `cout << oct << zahl << endl;` Funktionsnamen, also Zeiger auf eine Funktion, übergeben. Der Operator führt diese Funktionen aus, sodass die Einstellung auf die oktale Zahlenbasis beziehungsweise die Ausgabe einer neuen Zeile bewirkt werden. Die uns nicht vorliegende Implementierung zum Beispiel von `endl()` könnte wie folgt aussehen:

```
ostream& endl(ostream& os) {
   os.put('\n');
   os.flush();
   return os;
}
```

Der Ausgabeoperator ruft die übergebene Funktion auf:

```
ostream& ostream::operator<<(ostream& (*fp)(ostream&)) {
    *fp(*this);            // Funktionsaufruf
    return *this;
}
```

Die Anweisung

```
cout << zahl << endl;
```

wird ausgewertet zu

```
(cout.operator<<(zahl)).operator<<(endl);
```

In Kenntnis dieses Mechanismus können Sie nun selbst die tollsten Manipulatoren schreiben! Aber ehe Sie loslegen, schauen Sie sich erst die bereits vorhandenen an (Tabellen 10.3 bis 10.5), von denen einige in den Headern `<ios>` und `<iostream>` deklariert sind, andere (die mit Argumenten) jedoch in `<iomanip>` (iomanip = *input output manipulator*). Einschließen von `<iostream>` impliziert Inkludieren von `<ios>`.

Tabelle 10.3: `ios`-Manipulatoren

Name	Bedeutung
`boolalpha`	`true/false` alphabetisch ausgeben oder lesen
`noboolalpha`	`true/false` numerisch (1/0) ausgeben oder lesen
`showbase`	Basis anzeigen
`noshowbase`	keine Basis anzeigen
`showpoint`	nachfolgende Nullen ausgeben
`noshowpoint`	keine nachfolgenden Nullen ausgeben
`showpos`	+ bei positiven Zahlen anzeigen
`nowshowpos`	kein + bei positiven Zahlen anzeigen
`skipws`	Zwischenraumzeichen ignorieren
`noskipws`	Zwischenraumzeichen berücksichtigen
`uppercase`	E,X statt e,x
`nouppercase`	e,x statt E,X
`unitbuf`	Puffer nach jeder Ausgabe leeren
`nounitbuf`	Ausgabe puffern
`adjustfield:`	
`internal`	zwischen Vorzeichen und Wert auffüllen
`left`	linksbündige Ausgabe
`right`	rechtsbündige Ausgabe
`basefield:`	
`dec`	dezimal
`oct`	oktal
`hex`	hexadezimal
`floatfield:`	
`fixed`	Gleitkomma-Format
`scientific`	Exponential-Format

Tabelle 10.4: `iostream`-Manipulatoren

Name	Bedeutung	Typ
`endl`	neue Zeile ausgeben	`ostream&`
`ends`	Nullzeichen (`'\0'`) ausgeben	`ostream&`
`flush`	Puffer leeren	`ostream&`
`ws`	Zwischenraumzeichen aus der Eingabe entfernen	`istream&`

Manipulatoren sind sehr einfach anzuwenden – auch wenn die Erklärung ihrer Wirkungsweise vielleicht nicht so einfach wie die Anwendung ist. Die Beispiele mit `cout.precision()` und `cout.width()` können umgeschrieben werden:

Tabelle 10.5: `iomanip`-Manipulatoren

Name	Bedeutung
`resetiosflags(ios::fmtflags M)`	Flags entsprechend der Bitmaske `M` zurücksetzen
`setiosflags(ios::fmtflags M)`	Flags entsprechend `M` setzen
`setbase(int B)`	Basis 8, 10 oder 16 definieren
`setfill(char c)`	Füllzeichen festlegen
`setprecision(int n)`	Fließkommaformat (siehe Seite 380)
`setw(int w)`	Weite setzen (entspricht `width()`)

```
cout << setw(6) << setfill('0') << 999 << ' ';
cout << setprecision(8) << 1234.56789 << endl;
```

Das Ergebnis ist: *000999 1234.5679* (Annahme: `fixed` und `scientific` sind nicht gesetzt). Ist Ihnen etwas aufgefallen? Wie die Syntax zeigt, ist ein Manipulator *mit* Argument(en) kein Zeiger auf eine Funktion, sondern ein Funktionsaufruf. Die Autoren der *iostream*-Bibliothek konnten den Operator << nicht für alle nur denkbaren Fälle von Funktionen mit Parametern überladen. Daher wurde folgender Ausweg gewählt: Die aufgerufene Funktion muss ein Objekt einer (compilerspezifischen) Klasse, zum Beispiel `omanip` genannt, zurückgeben, das vom Ausgabeoperator verarbeitet werden kann. Im Header `<iomanip>` ist die Klasse `omanip` und ein `friend`-Operator << etwa der folgenden oder einer ähnlichen Art zu finden:

```
template<typename T>
class omanip {
     ostream& (*funktPtr)(ostream&, T);
     T arg;
   public:
     omanip(ostream& (*f)(ostream&, T), T obj)
     : funktPtr(f),
       arg(obj) {
     }
     friend ostream& operator<<(ostream&, omanip<T>&);
};

template<typename T>
ostream& operator<<(ostream& s, const omanip<T>& fobj) {
   return(*fobj.funktPtr)(s, fobj.arg);
}
```

Ein `omanip`-Objekt hat zwei private Variable: `funktPtr` ist ein Zeiger auf eine Funktion, die eine Referenz auf einen `ostream` zurückgibt und einen Parameter des Typs `ostream&` sowie ein Objekt des Typs `T` erwartet. In der Regel wird das Objekt vom Typ `int` oder `char` sein. Die zweite Variable `arg` enthält das Objekt vom Typ `T`.

Der Konstruktor initialisiert beide Variablen, die ihm als Parameter übergeben werden. Ferner finden Sie in `<iomanip>` eine Funktion (zum Beispiel) `setprecision()`. Die Funktion `setprecision()` gibt ein Objekt vom Typ `omanip` zurück:

```
omanip<int> setprecision(int p) {
   return omanip<int>(precision, p);
}
```

<int> rührt daher, dass die Klasse omanip als Template deklariert ist, um Manipulatorfunktionen mit verschiedenen Parametertypen zu erlauben. Dem Konstruktor eines omanip-Objekts wird die Adresse einer Funktion (precision) und der Wert p übergeben. Bei der Konstruktion dieses Objekts anlässlich der return-Anweisung werden diese Daten als Elementdaten abgelegt. Der Operator << ruft die im omanip-Objekt referenzierte Funktion auf, die wiederum die ostream-Funktion precision() aufruft, die uns ja schon bekannt ist (Seite 380). Solcherart Objekte werden auch *Funktionsobjekte* genannt, eine Variante der in Abschnitt 9.6 beschriebenen Funktoren. Grund: Über den Weg der Erzeugung eines Objekts wird eine Funktion aufgerufen, wenn auch nicht mit operator()(), sondern innerhalb operator<<() über einen Funktionszeiger.

Entsprechend zu der Klasse omanip für die Ausgabe gibt es die Klasse imanip für die Eingabe und die Klasse smanip für ios-Funktionen. Diese Namen werden jedoch nicht vom C++-Standard vorgeschrieben.

10.3.1 Eigene Manipulatoren

Eigene Manipulatoren ohne Parameter

Dies ist der einfachste Fall. Es muss nur eine Funktion mit der passenden Schnittstelle geschrieben werden. Der Manipulator endl von Seite 383 ist ein gutes Beispiel dafür.

Eigene Manipulatoren mit Parametern

Es gibt zwei Wege, eigene Manipulatoren zu schreiben. Der eine Weg führt über den beschriebenen Ansatz: Funktionen, die ein omanip-Objekt zurückgeben und sich auf den dazugehörenden Ausgabeoperator verlassen. Dieser Weg hat den gravierenden Nachteil, dass man sich auf nicht standardisierte Klassennamen verlassen muss. Deswegen wird hier nur der zweite Weg, die Realisierung mit einem Funktor, vorgeschlagen. Die Header-Datei enthält die Definition des Manipulators:

Listing 10.1: Manipulator-Klasse Leerzeilen

```
// cppbuch/k10/manipula.h
#ifndef MANIPULA_H
#define MANIPULA_H
#include<iostream>

class Leerzeilen {
  public:
    Leerzeilen(int i = 1) : anzahl(i) {}
    std::ostream& operator()(std::ostream& os) const {
       for(int i = 0; i < anzahl; ++i) {
         os << '\n';
       }
       os.flush();
       return os;
    }

  private:
    int anzahl;
};
```

```
inline std::ostream& operator<<(std::ostream& os,
                                const Leerzeilen& leerz) {
   return leerz(os);    // Funktoraufruf
}
#endif  // MANIPULA_H
```

Ein Funktor ist ein Objekt, das wie eine Funktion behandelt werden kann, wie auf Seite 344 gezeigt. Das Objekt kann beliebige Daten mit sich tragen. Dazu benötigt man nur noch einen mit der Klasse des Funktors überladenen Ausgabeoperator. Dies soll an dem einfachen Beispiel eines Manipulators `Leerzeilen(int z)`, der `z` Leerzeilen ausgibt, gezeigt werden. Eine mögliche Anwendung:

Listing 10.2: Anwendung des Manipulators

```
// cppbuch/k10/manipula.cpp
#include"manipula.h"
int main() {
   std::cout << Leerzeilen(25)         // Konstruktoraufruf!
             << "Ende" << std::endl;
}
```

würde den Bildschirm durch Ausgabe von 25 Leerzeilen löschen und dann den Text *Ende* ausgeben. Der Ablauf im `main()`-Program umfasst mehrere Schritte:

1. Zunächst wird ein Objekt vom Typ `Leerzeilen` konstruiert, das mit 25 als Parameter initialisiert wird.
2. Der Ausdruck `cout << Leerzeilen(25)` wird vom Compiler in die Langform `operator<<(cout, Leerzeilen(25))` umgewandelt. Daran ist zu sehen, dass das erzeugte Objekt als Parameter an den überladenen Operator weitergereicht wird.
3. Innerhalb des Ausgabeoperators wird der Funktionsoperator der Klasse `Leerzeilen` aufgerufen. Die Schreibweise `leerz(os)` wird vom Compiler zu `leerz.operator()(os)` umgewandelt. Damit ist der Ausgabestrom (hier `cout`) innerhalb der Operatorfunktion bekannt, und es kann die gewünschte Zahl von Leerzeilen ausgegeben werden.
4. Durch das per Referenz zurückgegebene `ostream`-Objekt ist eine Verkettung mit weiteren Operatoren denkbar.

Der Vorteil des Einsatzes von Funktoren liegt in der Einfachheit und darin, dass bei entsprechender Gestaltung eine beliebige Zahl von Parametern möglich ist.

10.4 Fehlerbehandlung

Bei der Ein- und Ausgabe können natürlich Fehler auftreten. Zur Erkennung und Behandlung von Fehlern stehen verschiedene Funktionen zur Verfügung. Ein Fehler wird durch das Setzen eines Status-Bits markiert, das durch den Aufzählungstyp `iostate` der Klasse `ios_base` (und damit auch `ios`) definiert wird. Die tatsächlichen Bitwerte sind implementationsabhängig.

```
enum iostate {
    goodbit   = 0x00,    // alles ok
    eofbit    = 0x01,    // Ende des Streams
    failbit   = 0x02,    // letzte Ein-/Ausgabe war fehlerhaft
    badbit    = 0x04     // ungültige Operation, grober Fehler
};
```

Der Stream ist nicht mehr benutzbar, falls `badbit` gesetzt ist. Tabelle 10.6 zeigt die Elementfunktionen, die auf die Statusbits zugreifen.

Tabelle 10.6: Abfrage des Ein-/Ausgabestatus

Funktion	Ergebnis
`iostate rdstate()`	aktueller Status
`bool good()`	wahr, falls »gut«, d.h. `rdstate() == 0`
`bool eof()`	wahr, falls Dateiende
`bool fail()`	wahr, falls `failbit` oder `badbit` gesetzt
`bool bad()`	wahr, falls `badbit` gesetzt
`void clear()`	Status auf `goodbit` setzen
`void clear(iostate s)`	Status auf `s` setzen
`void setstate(iostate)`	einzelne Statusbits setzen

Das folgende Demonstrationsprogramm zeigt die Anwendung der Funktionen zur Diagnose und Behebung von Syntaxfehlern beim Einlesen von `int`-Zahlen. Es wird dabei angenommen, dass beliebig oft `int`-Zahlen `i` eingelesen und angezeigt werden sollen, wobei vorher auftretende falsche Zeichen zu ignorieren sind.

Listing 10.3: `istream`-Status abfragen

```
// cppbuch/k10/iostate.cpp
#include<iostream>

using namespace std;

int main() {
    int i;
    ios::iostate status;

    while(true) {        // Schleifenabbruch mit break
        cout << "Zahl (Strg+D oder Strg+Z = Ende):";
        cin >> i;
        status = cin.rdstate();
        // Ausgabe der Statusbits
        cout << "status = " << status << endl;
        cout << "good() = " << cin.good() << endl;
        cout << "eof()  = " << cin.eof() << endl;
        cout << "fail() = " << cin.fail() << endl;
        cout << "bad()  = " << cin.bad() << endl;
        if(cin.eof())
            break;               // Abbruch
        // Fehlerbehandlung bzw. Ausgabe
        if(status) {
            cin.clear();      // Fehlerbits zurücksetzen
            cin.get();        // ggf. fehlerhaftes Zeichen entfernen
```

```
        }
        else cout << "*** " << i << endl;
    }
}
```

Die Funktion `void clear(iostate statuswort = goodbit)` erlaubt es, den Status zu setzen. Beispielsweise kann das `badbit` bei Erhaltung der anderen Bits mit `cin.clear(ios::badbit | cin.rdstate())` gesetzt werden. Durch Eingabe von Buchstaben ist die Syntax von `int`-Zahlen nicht erfüllt, wie mit `fail()` angezeigt wird. Die Tastenkombination Strg+Z oder Strg+D liefert einen Wert ungleich 0 für `eof()` und führt zum Verlassen der Schleife.

Exception ios::failure

Wenn seitens des Ein-/Ausgabesystems ein Fehler gefunden und deswegen intern die Funktion `setstate(failbit)` aufgerufen wird, *kann* es sein, dass sie eine `ios_base::failure`-Exception wirft. Der Standard legt sich nicht fest, weil diese Exception neu aufgenommen wurde und das Verhalten bisher gültiger Programme sich nicht ändern soll – die Entscheidung liegt beim Hersteller des Compilers. Da die Klasse `ios` von `ios_base` erbt, kann sie kürzer `ios::failure` genannt werden. `ios::failure` erbt von `system_error`. Man kann sie bei eigenen Programmen selbst werfen und auswerten. Das Programm auf Seite 636 zeigt eine mögliche Anwendung.

10.5 Typumwandlung von Dateiobjekten nach bool

Die Abfrage, ob eine Datei geöffnet werden kann, wird über eine `if`-Abfrage etwa der folgenden Art gelöst:

```
ifstream quellfile;
quellfile.open("text.dat");
if(!quellfile) {
    cerr << "Datei kann nicht geöffnet werden!";
    exit(-1);
}
```

Ob das Ende einer Datei erreicht worden ist, kann in einer Bedingung wie folgt festgestellt werden:

```
while(quellfile.get(c)) {
    zielfile.put(c);
}
```

Wie funktioniert dieser geheimnisvolle Mechanismus? Erinnern wir uns daran, dass der Compiler automatisch versucht, einen nicht ganz passenden Datentyp in einen passenden umzuwandeln, wenn es nötig sein sollte; der Versuch gelingt nicht immer. Dazu kann

er Typumwandlungskonstruktoren (siehe Seite 160) und eingebaute oder selbst definierte Typumwandlungsoperatoren (siehe Seite 337) benutzen. Zur Klasse `ios` gibt es einen vordefinierten Typumwandlungsoperator, der ein Dateiobjekt oder eine Referenz darauf in einen Wert vom Typ `void*` umwandelt. Zusätzlich wird der Negationsoperator überladen. Diese Operatoren werden vom Compiler benutzt, um die Bedingung auswerten zu können. Sie benutzen die Funktion `fail()` zum Lesen des Status und sind etwa wie folgt implementiert:

```
// Konversion eines Objekts in einen void-Zeiger
ios::operator void *() const {
   if(fail()) return static_cast<void*>(0);
   else       return static_cast<void*>(this);
}

bool ios::operator!() const { // überladener Negationsoperator
   return fail();
}
```

Die Anweisung `while(cin) cin.get(c);` wird interpretiert als

```
while(cin.operator void*())
  cin.get(c);
```

und die Anweisung `if(!cin.get(c)) {...}` entspricht

```
if((cin.get(c)).operator!()) {
  //...
}
```

10.6 Arbeit mit Dateien

Die Arbeit mit Dateien ist teilweise aus Kapitel 2 bekannt. In diesem Abschnitt finden sich einige Ergänzungen. Zum Öffnen einer Datei wird ein bestimmter Modus angegeben (englisch *openmode*) wie zum Beispiel `ios::binary`. Die Tabelle 10.7 zeigt die möglichen Werte.

Tabelle 10.7: `ios_base`-Öffnungsarten für Ströme

Modus	Bedeutung
`app`	beim Schreiben Daten an die Datei anhängen
`ate`	nach dem Öffnen an das Dateiende springen
`binary`	keine Umwandlung verschiedener Zeilenendekennungen
`in`	zur Eingabe öffnen
`out`	zur Ausgabe öffnen
`trunc`	vorherigen Inhalt der Datei löschen

Bestimmte Werte sind Stream-abhängig voreingestellt. Beispiel: Bei einem `ifstream`-Objekt ist `ios::in` voreingestellt. Die Angabe eines Modus überschreibt den vorgegebenen

Modus! Deswegen sollte `ios::binary` bei `ifstream`-Objekten mit `ios::in` gekoppelt werden, d.h. Modus = `ios::binary|ios::in` (`ofstream` entsprechend mit `ios::out`). Die verschiedenen Öffnungsarten können durch Verknüpfung mit dem Oder-Operator kombiniert werden. Tabelle 10.8 zeigt die sinnvollen möglichen Kombinationen. Ein Beispiel:

```
// nur am Ende schreiben
ofstream Ausgabestrom("Ausgabe.dat", ios::out | ios::app);
// Lesen und Schreiben (Beispiel folgt auf Seite 392):
fstream EinAusgabestrom("Datei.txt", ios::in | ios::out);
```

Tabelle 10.8: Kombinationen der Dateiöffnungsarten

binary	in	out	trunc	app / ate
	•			
	•	•		
	•	•	•	
		•		
		•	•	
		•		•
•	•			
•	•	•		
•	•	•	•	
•		•		
•		•	•	
•		•		•

10.6.1 Positionierung in Dateien

Manchmal ist es wünschenswert, eine Datei nicht nur sequenziell zu lesen oder zu schreiben, sondern die Position frei zu bestimmen. Dazu gehört auch, die aktuelle Position zu ermitteln. Die Tabelle 10.9 zeigt die vorhandenen Funktionen. Die Endung »g« steht für »get« (= zu lesende Datei) und »p« steht für »put« (= zu schreibende Datei). Die Bezugsposition `Bezug` in Tabelle 10.9 kann einen von drei möglichen Werten annehmen:

- `ios::beg` relativ zum Dateianfang
- `ios::cur` relativ zur aktuellen (englisch *current*) Position
- `ios::end` relativ zum Dateiende

Das folgende Programmfragment zeigt eine Anwendung:

Tabelle 10.9: Ermitteln und Suchen von Dateipositionen

Rückgabetyp	Funktion	Bedeutung
`ios::pos_type`	`tellg()`	aktuelle Leseposition
`ios::pos_type`	`tellp()`	aktuelle Schreibposition
`istream&`	`seekg(p)`	absolute Position `p` aufsuchen
`istream&`	`seekg(r, Bezug)`	relative Position `r` aufsuchen (zur Bezugsposition siehe Text)
`ostream&`	`seekp(p)`	absolute Position `p` aufsuchen
`ostream&`	`seekp(r, Bezug)`	relative Position `r` aufsuchen

```
// Auszug aus cppbuch/k10/seektell.cpp
ifstream einIfstream;
```

```
einIfstream.open("seek.dat", ios::binary|ios::in);
einIfstream.seekg(9);   // absolute Leseposition 9 suchen (Zählung ab 0)
char c;
einIfstream.get(c);     // an Pos. 9 lesen, get schaltet Position um 1 weiter
einIfstream.seekg(2, ios::cur); // 2 Positionen weitergehen
ios::pos_type position = einIfstream.tellg();// akt. Position merken
// ...
einIfstream.seekg(-4, ios::end);   // 4 Positionen vor dem Ende
// ...
einIfstream.seekg(position, ios::beg); // zur gemerkten Position gehen
```

Daraus ergibt sich, dass die relative Positionierung zum Dateianfang redundant ist, d. h. `seekg(x, ios::beg)` ist dasselbe wie `seekg(x)`;

10.6.2 Lesen und Schreiben in derselben Datei

Wenn eine Datei als Datenbasis genutzt wird, ist es interessant, Daten zu lesen und geänderte Daten in *derselben* Datei zu aktualisieren. Für diesen Zweck gibt es die Klasse `fstream`, die von der Klasse `iostream` erbt, die wiederum von den Klassen `istream` und `ostream` abgeleitet ist. Damit hat `fstream` die Eigenschaft, sowohl für die Ein- als auch für die Ausgabe geeignet zu sein. Alle Klassen, die mit Dateien arbeiten, benutzen Puffer. In `fstream` wird derselbe Pufferspeicher zum Lesen und zum Schreiben benutzt. Ein einfaches Programm zeigt die Nutzung eines `fstream`s, bei dem die Funktionen zum Aufsuchen von Positionen naturgemäß eine starke Rolle spielen.

Listing 10.4: Lesen/Schreiben derselben Datei

```
// cppbuch/k10/fstream2.cpp
#include<fstream>
#include<iostream>
using namespace std;

int main() {  // Lesen und Schreiben derselben Datei
   // Datei anlegen
   fstream filestream("fstream2.dat", ios::out | ios::trunc);
   filestream.close();          // leere Datei existiert jetzt
   int i;                       // Hilfsvariable
   // Datei zum Lesen und Schreiben öffnen:
   filestream.open("fstream2.dat", ios::in | ios::out);

   // schreiben:
   for(i = 0; i < 20; ++i) {
      filestream << i << ' ';       // kein EOF direkt nach der letzten Zahl
   }
   filestream << endl;

   // lesen:
   filestream.seekg(0);             // Anfang suchen
   while(filestream.good()) {
      filestream >> i;              // lesen
      if(filestream.good()) {
         cout << i << ' ';          // Kontrollausgabe
      }
```

```
      else {
         cout << endl << "Dateiende erreicht (oder Lesefehler)";
      }
   }
   cout << endl;  // Ergebnis: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
   filestream.clear();              // EOF-Status löschen

   // Inhalt teilweise überschreiben
   filestream.seekp(5);             // Position 5 zum Schreiben suchen
   filestream << "neuer Text ";     // ab Pos. 5 überschreiben
   filestream.seekg(0);             // Anfang zum Lesen suchen
   char buf[100];
   filestream.getline(buf,100);     // Zeile lesen
   cout << buf << endl;      // Kontrollausgabe. Ergebnis:
                        // 0 1 2 neuer Text 9 10 11 12 13 14 15 16 17 18 19
}
```

10.7 Umleitung auf Strings

Die Ausgabe kann mithilfe von `ostringstream`-Objekten (*output string stream*) auf Strings umgeleitet werden. Dies ist dann sinnvoll, wenn die Ausgabe nicht unmittelbar erfolgen soll. Ähnliches gilt auch für das Lesen aus Strings anstelle einer Datei (`istringstream`). `stringstream`-Klassen sind im Header `<sstream>` deklariert. Hier sei beispielhaft nur die Ausgabe behandelt. Es ist eine Funktion `ZahlToString()` angegeben, die eine Zahl in einen formatierten String umwandelt. Der String kann anschließend anderweitig ausgewertet oder ausgegeben werden.

Die Funktion ist für verschiedene Typen von Zahlen geeignet. Die Parameter zur Formatierung können weggelassen werden, wenn das vorgegebenen Format ausreicht. Ein Beispiel zeigt Anwendung und Funktionsweise.

Listing 10.5: Stream-Ausgabe in String umleiten

```
// cppbuch/k10/strstrea.cpp
#include<iostream>
#include"zahlToString.h"
using namespace std;

int main() {                            // Anwendung
  double xd = 73.1635435363;            // Ausgabe:
  cout << zahlToString(xd) << endl;     //73.1635
  cout << zahlToString(xd,12) << endl; //      73.1635
  cout << zahlToString(xd,12,1) << endl; //   7.3164e+01
  cout << zahlToString(xd,12,1,3) << endl; //    7.316e+01
  cout << zahlToString(xd,12,0,3) << endl; //       73.164
  int xi = 1234567;
  cout << zahlToString(xi) << endl;     //1234567
  cout << zahlToString(xi,14) << endl; //        1234567
```

```
  float xf = 1234.567;
  cout << zahlToString(xf) << endl;    //1234.57
  unsigned long xl = 123456789L;
  cout << zahlToString(xl) << endl;    //123456789
}
```

Listing 10.6: Prototypen

```
// cppbuch/k10/zahlToString.h
#ifndef ZAHLTOSTRING_H
#define ZAHLTOSTRING_H
#include<string>

std::string zahlToString(double d, unsigned int weite = 0,
    unsigned int format = 2, // 0: fix, 1: scientific, sonst: automatisch
    unsigned int anzahlNachkommastellen = 4); // nur format 0/1
// ganze Zahlen:
std::string zahlToString(long i,     unsigned int weite = 0);
std::string zahlToString(int i,      unsigned int weite = 0);
std::string zahlToString(unsigned long i, unsigned int weite = 0);
std::string zahlToString(unsigned int i, unsigned int weite = 0);
#endif
```

Listing 10.7: Implementierung

```
// cppbuch/k10/zahlToString.cpp
#include<sstream>
#include<iostream>
#include"zahlToString.h"

// Das ostringstream-Objekt wandler in der folgenden Funktion
// zahlToString(double, unsigned int, unsigned int, unsigned int)
// stellt dynamisch Platz bereit, ohne dass man sich besonders darum kümmern muss. Der
// Aufruf wandler.str(); gibt alle in den Stringstream geschriebenen Ausgaben als String zurück.
std::string zahlToString(double d, unsigned int weite,
                         unsigned int format, // 0: fix, 1: scientific,
                                              // sonst: automatisch
                         unsigned int anzahlNachkommastellen ) { // nur Format 0/1
  std::ostringstream wandler;
  if(format == 0) {
    wandler.setf(std::ios::fixed, std::ios::floatfield);
    if(anzahlNachkommastellen > 0) {
      wandler.setf(std::ios::showpoint);
    }
  }
  else {
    if(format == 1) {
      wandler.setf(std::ios::scientific, std::ios::floatfield);
    }
  }
  if(format == 0 || format == 1) {
    wandler.precision(anzahlNachkommastellen);
  }
```

```
   if(weite > 0) {
      wandler.width(weite);
   }
   // Zahl und abschließendes Nullzeichen in den Strom einfügen
   wandler << d << std::ends;
   return wandler.str();
}

std::string zahlToString(long i, unsigned int weite) {
   return zahlToString(static_cast<double>(i), weite, 0, 0);
}
std::string zahlToString(int i, unsigned int weite) {
   return zahlToString(static_cast<double>(i), weite, 0, 0);
}
std::string zahlToString(unsigned long i, unsigned int weite) {
   return zahlToString(static_cast<double>(i), weite, 0, 0);
}
std::string zahlToString(unsigned int i, unsigned int weite) {
   return zahlToString(static_cast<double>(i), weite, 0, 0);
}
```

Mit einem `istringstream`-Objekt kann aus einem String gelesen werden, ein `stringstream`-Objekt erlaubt Lesen und Schreiben.

10.8 Ergänzungen

locale-Objekt ermitteln

Das für einen Stream gültige `locale`-Objekt (`locale` siehe Seite 821 ff.) für die Sprachumgebung kann mit `getloc()` ermittelt werden, z. B. `cout.getloc();`.

Streams verbinden mit tie()

Die Funktion `tie()` der Klasse `basic_ios` verbindet und löst Verbindungen zwischen einem `istream` und einem `ostream`. Wenn das Programmfragment

```
cout << "Zahl eingeben: ";
cin >> Zahl;
```

gegeben ist, dann ist nicht sichergestellt, dass die Bildschirmausgabe *vor* der Eingabeoperation erscheint, weil der Ausgabepuffer noch nicht voll ist. Auf Seite 379 wird das Problem mit dem Setzen von `ios::unitbuf` gelöst, eine andere Möglichkeit ist das Verbinden der Streams:

```
cin.tie(&cout);  // Verbindung herstellen
cout << "Zahl eingeben: ";
cin >> Zahl;       // Diese Zeile verursacht jetzt cout.flush()
// Abfrage der Verbindung
if(cin.tie() == &cout) {
```

```
  cout << "Streams cin und cout sind verbunden" << endl;
}
cin.tie(0);        // Verbindung lösen
```

sentry

Es gibt sowohl die Klasse `basic_istream::sentry` als auch die Klasse `basic_ostream::sentry`. `sentry`-Objekte sorgen für eine sichere Umgebung bei Ein- bzw. Ausgabeoperationen. Der Konstruktor enthält Code, der vorher ausgeführt werden soll (Präfix-Code), zum Beispiel den Puffer eines mit `tie()` verbundenen Streams schreibend zu leeren (`flush()`). Der `bool`-Operator gibt an, ob der Stream in einem guten Zustand ist. Der Destruktor enthält Code, der nach der Operation ausgeführt werden soll (Suffix-Code), zum Beispiel das Werfen einer Exception, wenn der Stream in einen Fehlerzustand gelangt ist. Anwendungsbeispiele sind auf den Seiten 835 und 836 zu finden. Ein typisches Anwendungsmuster ist:

```
// Beispiel für istream::sentry
void f(istream& is) {
  istream::sentry s(is); // Präfix-Code
  if(s) {               // ok? (bool sentry::operator())
    // irgendetwas mit is tun
  }
}                       // Suffix-Code (Destruktor)
```

Mit `sentry`-Objekten wird das C++-Prinzip »Resource Acquisition Is Initialization« (RAII, siehe Glossar) realisiert.