WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE

it
informatik

Bjarne Stroustrup

Einführung in die Programmierung mit C++

PEARSON
Studium

10.8 Benutzerdefinierte Ausgabeoperatoren

Den Ausgabeoperator << für einen gegebenen Typ zu definieren, ist eine mehr oder weniger triviale Aufgabe. Das größte Problem besteht üblicherweise in der Festlegung des Ausgabeformats, da verschiedene Leute durchaus unterschiedliche Vorstellungen davon haben können, wie die Ausgabe aussehen sollte. Doch auch wenn sich kein einzelnes Ausgabeformat findet, das allen Anwendungsfällen gerecht werden kann, ist es meist von Vorteil, für einen benutzerdefinierten Typ den <<-Operator zu definieren, damit wir zumindest während des Debuggens und der ersten Entwicklungsphasen eine einfache Möglichkeit haben, Objekte des Typs auszugeben. Später können wir uns dann überlegen, einen ausgefeilteren <<-Operator zur Verfügung zu stellen, dem der Benutzer Formatierungshinweise übergeben kann. Im Übrigen steht es Benutzern des Typs natürlich frei, in Situationen, wo die vom <<-Operator erzeugte Ausgabe ungeeignet erscheint, den Operator zu übergehen und die einzelnen Teile des benutzerdefinierten Typs so auszugeben, wie es die Anwendung erfordert.

Der folgende Code definiert einen einfachen Ausgabeoperator für den Typ **Date** aus §9.8, der Jahr, Monat und Tag in runden Klammern und getrennt durch Kommata ausgibt:

```
ostream& operator<<(ostream& os, const Date& d)
{
  return os << '(' << d.year()
    << ',' << d.month()
    << ',' << d.day() << ')';
}
```

Dieser Operator gibt den 30. August 2004 als „(2004,8,30)“ aus. Auf diese Form der Darstellung (einfache elementweise Auflistung) greifen wir immer dann zurück, wenn wir einen Typ mit wenigen Membern vorliegen haben und es keine Ideen oder Gründe für eine andere Darstellung gibt.

In §9.6 ist bereits angeklungen, dass für benutzerdefinierte Operatoren die zugehörige Funktion aufgerufen wird. Wie dies geht, können wir uns hier an einem Beispiel ansehen. Wenn << für **Date** definiert wurde und **d1** ein Objekt vom Typ **Date** ist, steht

```
cout << d1;
```

für den Aufruf:

```
operator<<(cout,d1);
```

Beachten Sie, dass **operator<<()** als erstes Argument eine **ostream**-Referenz übernimmt, die sie als Rückgabewert wieder zurückliefert. Dieses Weiterreichen des Ausgabestreams sorgt dafür, dass wir Ausgabeoperationen aneinanderreihen können. Beispielsweise können wir zwei Datumsangaben wie folgt hintereinander ausgeben:

```
cout << d1 << d2;
```

Für diese Verkettung wird zunächst die erste <<- und dann die zweite <<-Operation ausgeführt:

```
cout << d1 << d2;  // bedeutet operator<<(cout,d1) << d2;
                   // bedeutet operator<<(operator<<(cout,d1),d2);
```

Konkret bedeutet dies: Gib zunächst **d1** nach **cout** aus und schreibe dann **d2** in den Ausgabestream, der das Ergebnis der ersten Ausgabeoperation darstellt. Technisch gesehen können wir jede der drei im letzten Beispiel angegebenen Varianten zur Ausgabe von **d1** und **d2** verwenden, aber wir wählen natürlich die Variante, die am einfachsten zu lesen ist.

10.9 Benutzerdefinierte Eingabeoperatoren

Den Eingabeoperator **>>** für einen gegebenen Typ und ein Eingabeformat zu definieren, ist im Wesentlichen eine Übung in korrekter Fehlerbehandlung – und kann daher ziemlich knifflig werden.

Der folgende Code definiert einen einfachen Eingabeoperator für die **Date**-Klasse aus §9.8, der Datumsangaben liest, die in dem gleichen Format geschrieben sind, das auch der oben definierte **<<**-Operator benutzt.

```
istream& operator>>(istream& is, Date& dd)
{
  int y, m, d;
  char ch1, ch2, ch3, ch4;
  is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
  if (!is) return is;
  if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') {   // Hoppla: Formatfehler
    is.clear(ios_base::failbit);
    return is;
  }
  dd = Date(y,Date::Month(m),d);  // aktualisiere dd
  return is;
}
```

Dieser **>>**-Operator liest Eingaben der Form „(2004,8,20)“ und versucht aus den drei Integer-Werten ein **Date**-Objekt zu erzeugen. Wie üblich ist die Eingabe schwerer zu handhaben als die Ausgabe. Bei der Eingabe kann einfach mehr schiefgehen als bei der Ausgabe (und häufig wird es dies auch).

Wenn der **>>**-Operator keine Eingabe im Format (*Integer*,*Integer*,*Integer*) vorfindet, setzt er den Stream in einen der Zustände **fail**, **eof** oder **bad** und lässt das Ziel der Operation, sein **Date**-Argument, unverändert. Die Memberfunktion **clear()** wird dazu verwendet, den Status des **istream**-Streams festzulegen (wobei das **ios_base::failbit**-Argument offensichtlich die Aufgabe hat, den Stream in den **fail()**-Zustand zu versetzen). Sehr begrüßenswert ist, dass die Operatorfunktion das Zielobjekt **dd** im Falle eines Scheiterns unverändert lässt; das hilft den Benutzern des Operators, sauberen, klaren Code zu schreiben. Ideal wäre es, wenn die **operator>>()**-Funktion zudem keine Zeichen konsumieren (verwerfen) würde, die sie nicht weiterverarbeitet. Doch angesichts der Tatsache, dass wir unter Umständen bereits eine ganze Reihe von Zeichen eingelesen haben, bevor wir einen Formatfehler abfangen, ist dies im vorliegenden Fall zu schwierig zu implementieren. Nehmen Sie zum Beispiel die Eingabe „(2004,8,30}“. Erst wenn wir die abschließende }-Klammer sehen, wissen wir, dass ein Formatfehler vorliegt. Dann ist es allerdings zu spät, um die vielen eingelesenen Zeichen wieder zurückzustellen, denn laut Standard ist lediglich das korrekte Zurückstellen des jeweils letzten Zeichens mit **unget()** garantiert. Wenn **operator>>()** ein ungültiges Datum wie z.B. „(2004,8,32)“ einliest, wirft der **Date**-Konstruktor eine Ausnahme, die uns aus dem **operator>>()**-Aufruf führt.

10.10 Standardlösung für eine Einleseschleife

In §10.5 haben wir gesehen, wie wir Dateien lesen und schreiben können. Das war allerdings, bevor wir uns eingehender mit der Fehlerbehandlung beschäftigt haben (§10.6), sodass wir der Einfachheit halber davon ausgegangen sind, dass wir die Datei ungestört vom Anfang bis zum Ende einlesen können. Angesichts des Umstands, dass wir nicht selten separate Tests durchführen, um sicherzustellen, dass eine Datei gültig ist, kann diese Annahme durchaus statthaft und vernünftig sein. Auf der anderen Seite gibt es immer wieder Situationen, in denen wir die eingelesenen Daten noch während des Einlesens kontrollieren möchten. Eine allgemeine Strategie hierzu sieht wie folgt aus (**istr** sei ein **istream**-Stream):

```
My_type var;
while (istr>>var) {    // lies bis zum Dateiende
  // evtl. prüfen, ob var gültig ist
  // etwas mit var tun
}
// die Wiederaufnahme nach einem Wechsel in den bad-Zustand ist selten möglich
// und sollte nur im absoluten Notfall versucht werden
if (istr.bad()) error("Fehler in Eingabestream");
if (istr.fail()) {
  // was ist ein gültiges Abschlusszeichen?
}
// weitermachen: das Dateiende wurde erreicht
```

Wir lesen eine Folge von Werten in Variablen ein. Wenn wir keine weiteren Werte einlesen können, prüfen wir den Streamstatus um festzustellen, warum das Einlesen beendet wurde. Wie in §10.6 können wir den Code ein wenig verbessern, indem wir den **istream**-Stream eine Ausnahme vom Typ **failure** werfen lassen, wenn er in den **bad**-Zustand wechselt. Damit entfällt die Notwendigkeit den **bad**-Zustand jedes Mal explizit abzufragen.

```
// irgendwo: istr soll beim Wechseln in den bad-Zustand eine Ausnahme werfen
istr.exceptions(istr.exceptions() | ios_base::badbit);
```

Wir könnten uns auch entscheiden, ein spezielles Zeichen als Abschlusszeichen festzulegen:

```
My_type var;
while (istr>>var) {    // lies bis zum Dateiende
  // evtl. prüfen, ob var gültig ist
  // etwas mit var tun
}
if (istr.fail()) {    // verwende '|' als Abschlusszeichen und/oder Trennzeichen
  istr.clear();
  char ch;
  if (!(istr>>ch && ch=='|')) error("ungueltiger Abschluss der Eingabe");
}
// weitermachen: "End-of-File" oder Abschlusszeichen wurde erreicht
```

Wenn wir kein Abschlusszeichen, sondern nur das „End-of-File“-Signal als Ende der Eingabe akzeptieren möchten, müssen wir nur den Test vor dem Aufruf von **error()** entfernen. Abschlusszeichen sind allerdings äußerst hilfreich, wenn wir Dateien mit verschachtelten Datenkonstrukten einlesen, beispielsweise eine Datei mit monatlichen Messwerten, die tägliche Messwerte enthalten, die wiederum stündliche Messwerte enthalten und so weiter. Wir halten uns daher die Option eines Abschlusszeichens weiter offen.

Leider ist der Code immer noch etwas unübersichtlich. Besonders die wiederholte Überprüfung des Abschlusszeichens, die gerade, wenn wir viele Dateien einlesen, sehr lästig wird, ist uns ein Dorn im Auge. Durch Auslagerung des Codes in eine eigene Funktion können wir hier Abhilfe schaffen:

```
// irgendwo: der Stream soll beim Wechseln in den bad-Zustand eine Ausnahme werfen:
istr.exceptions(istr.exceptions() | ios_base::badbit);

void end_of_loop(istream& istr, char term, const string& message)
{
  if (istr.fail()) {    // verwende term als Abschluss- und/oder Trennzeichen
    istr.clear();
    char ch;
    if (istr>>ch && ch==term) return;   // alles ist okay
    error(message);
  }
}
```

Dies reduziert die Einleseschleife zu:

```
My_type var;
while (istr>>var) {    // lies bis zum Dateiende
  // evtl. prüfen, ob var gültig ist

  // etwas mit var tun
}
end_of_loop(istr,'|',"ungueltiger Abschluss der Datei");   // Test, ob wir fortfahren können

// weitermachen: "End-of-File" oder Abschlusszeichen wurde erreicht
```

Die Funktion **end_of_loop()** unternimmt nur dann etwas, wenn sich der Stream im **fail()**-Zustand befindet. Wir denken, dies ist einfach und unspezifisch genug, um die Funktion allgemein einsetzen zu können.

10.11 Eine strukturierte Datei lesen

In diesem Abschnitt wollen wir unsere Standardlösung an einem konkreten Beispiel testen und dabei – wie Sie es schon kennen – einige allgemein anwendbare Design- und Programmiertechniken illustrieren. Stellen Sie sich vor, Sie hätten eine Datei mit Temperaturablesungen vorliegen, die wie folgt strukturiert sind:

- Eine Datei besteht aus Jahreseinträgen (mit monatlichen Ablesungen).
  - Ein Jahreseintrag beginnt mit **{ year**, gefolgt von einer ganzen Zahl, die das Jahr angibt, also z.B. **1900**, und endet mit **}**.
- Ein Jahreseintrag beinhaltet Monatseinträge (mit täglichen Ablesungen).
  - Ein Monatseintrag beginnt mit **{ monat**, gefolgt von einem dreibuchstabigen Monatscode, also z.B. **jan**, und endet mit **}**.
- Ein Messwert besteht aus einer Zeitangabe und dem Temperaturwert.
  - Eine Messwert beginnt mit **(**, gefolgt vom Tag im Monat, der Stunde und dem Temperaturwert und endet mit **)**.

Zum Beispiel:

```
{ year 1990 }
{year 1991 { month jun }}
{ year 1992 { month jan ( 1 0 61.5) } {month feb (1 1 64) (2 2 65.2) } }
{year 2000
  { month feb (1 1 68 ) (2 3 66.66 ) ( 1 0 67.2)}
  {month dec (15 15 -9.2 ) (15 14 -8.8) (14 0 -2) }
}
```

Tipp

Dieses Format ist etwas absonderlich. Eine Eigenschaft, die es sich mit vielen anderen Dateiformaten teilt. Zwar gibt es in der Industrie Bestrebungen hin zu einheitlicheren, hierarchisch strukturieren Dateien (wie z.B. HTML- und XML-Dateien), doch die Realität sieht immer noch so aus, dass wir das Format der Dateien, die wir lesen sollen, nur selten beeinflussen können. Die Dateien sind so wie sie sind, und wir müssen sie lesen. Treffen wir dabei auf ein besonders schauriges Format oder auf Dateien, die zu viele Fehler enthalten, können wir ein Umformatierungsprogramm schreiben, das die Daten in ein Format umwandelt, welches unser Programm besser verarbeiten kann. Freie Hand haben wir dagegen zumeist bei der Entscheidung, wie wir die Daten auf geeignete Weise im Arbeitsspeicher ablegen, und bei der Wahl des Ausgabeformats, das wir nicht selten nach Bedarf und eigenen Vorlieben festlegen können.

Für die weiteren Ausführungen gehen wir davon aus, dass wir mit dem oben definierten Format für die Temperaturmesswerte leben müssen. Glücklicherweise enthält das Format einige selbstidentifizierende Komponenten wie die Jahres- und Monatseinträge (die an die Tags von HTML oder XML erinnern). Demgegenüber ist das Format der einzelnen Messwerte nur wenig hilfreich. Beispielsweise gibt es keinerlei Informationen, die uns weiterhelfen könnten, wenn die Werte für Tag und Stunde vertauscht sind oder jemand eine Datei mit Temperaturdaten in Celsius abgegeben hat, während das Programm Fahrenheit erwartet (oder umgekehrt). Wir müssen sehen, wie wir allein damit fertig werden.

10.11.1 Repräsentation im Speicher

Wie sollen wir die Daten im Speicher ablegen? Die naheliegende erste Wahl sind drei Klassen **Year**, **Month** und **Reading**, die exakt den Aufbau der Eingabe widerspiegeln. **Year** und **Month** sind von offensichtlichem Nutzen für die weitere Verarbeitung der Daten. Mit ihrer Hilfe können wir die Temperaturwerte verschiedener Jahre vergleichen, monatliche Durchschnittstemperaturen berechnen, die Monate eines Jahrs vergleichen, die Durchschnittstemperaturen eines bestimmten Monats im Verlauf der Jahre vergleichen, die Temperaturdaten mit Aufzeichnungen der Sonnenstunden und der Luftfeuchtigkeit abgleichen. Kurz gesagt, die Klassen **Year** und **Month** kommen der Art und Weise entgegen, wie wir mit Temperatur- und Wetterdaten arbeiten: **Month** verwahrt die Informationen eines Monats und **Year** die Informationen eines Jahres. Wie aber sieht es mit der Klasse **Reading** aus? **Reading** repräsentiert eine einzelne Ablesung, stellt also letzten Endes die Abstraktion eines technischen Gerätes (eines Sensors) dar. Die Daten einer Ablesung (Tag im Monat, Stunde, Temperatur) sind etwas „befremdend“ und ergeben nur im Kontext eines **Month**-Objekts einen Sinn. Außerdem sind sie ungeordnet, d.h., es gibt keinerlei Zusage, dass die Ablesungen nach dem Tag im Monat oder der Stunde der Ablesung geordnet sind. Bevor wir also irgendetwas Interessantes mit den Werten anfangen können, müssen wir sie zuerst einmal sortieren.

Zusätzlich gehen wir bei der Entscheidung, wie die Temperaturdaten im Speicher abzulegen sind, von folgenden Annahmen aus:

- Wenn es für einen Monat eine Ablesung gibt, liegen für diesen Monat meist noch weitere Ablesungen vor.
- Wenn es für einen Tag eine Ablesung gibt, liegen für diesen Tag meist noch weitere Ablesungen vor.

Dies vorausgesetzt ist es sinnvoll, ein **Year** als einen Vektor von 12 Monaten (**Month**-Objekten) darzustellen, einen **Month** als einen Vektor von ungefähr 30 Tagen (**Day**-Objekten) und einen **Day** als 24 Temperaturwerte (einen Wert für jede Stunde). Dies ist ein einfaches Design, leicht zu handhaben und vielseitig einsetzbar. **Day**, **Month** und **Year** sind einfache Datenstrukturen mit jeweils einem Konstruktor. Da wir planen, **Month**- und **Day**-Objekte als Teile eines **Year**-Objekts zu erzeugen, noch bevor wir wissen, welche Temperaturablesungen vorliegen, benötigen wir für die Stunden eines Tages, für die wir noch keine Daten eingelesen haben, etwas mit der Bedeutung „keine Ablesung“.

```
const int not_a_reading = -7777;   // Wert unterhalb der absoluten Nulltemperatur
```

Da uns überdies aufgefallen ist, dass es viele Monate gibt, für die keine Daten vorliegen, führen wir zur direkten Repräsentation dieses Fakts eine weitere symbolische Konstante mit der Bedeutung „kein Monat“ ein. Wir können dann sicher sein, dass es für den betreffenden Monat keine Daten gibt – ohne dass wir dazu die einzelnen Tage durchgehen müssen.

```
const int not_a_month = -1;
```

Die drei Schlüsselklassen lauten damit:

```
struct Day {
  vector<double> hour;
  Day(); // initialisiere die Stunden mit dem Wert für "keine Ablesung"
};

Day::Day()
  : hour(24)
{
  for (int i = 0; i<hour.size(); ++i) hour[i]=not_a_reading;
}

struct Month {   // Temperaturablesungen eines Monats
  int month;             // [0:11], 0 steht für Januar
  vector<Day> day;   // [1:31], pro Tag ein Vektor mit Ablesungen
  Month()                // max. 31 Tage in einem Monat (day[0] wird nicht genutzt)
    :month(not_a_month), day(32) { }
};

struct Year {   // Temperaturablesungen eines Jahres, organisiert nach Monaten
  int year;                       // positiv == n.Chr.
  vector<Month> month;      // [0:11], 0 steht für Januar
  Year() :month(12) { }          // 12 Monate in einem Jahr
};
```

Alle drei Klassen sind im Grunde einfache Vektoren ihrer „Teile“. **Month** und **Year** enthalten zudem Member, die sie identifizieren (**month** bzw. **year**).

Es gibt in diesem Code verschiedene „magische Konstanten“ (z.B. **24**, **32** und **12**). Grundsätzlich versuchen wir, solche literalen Konstanten im Code zu vermeiden. Die hier auftauchenden Konstanten sind ziemlich fundamental (die Anzahl Monate in einem Jahr ändert sich nur selten) und werden im Rest des Codes nicht mehr verwendet. Wir haben sie im Code belassen, um Sie noch einmal auf das Problem der „magische Konstanten“ aufmerksam zu machen: Symbolische Konstanten sind fast immer vorzuziehen (§7.6.1). Wenn **32** wie hier als Anzahl der Tage in einem Monat verwendet wird, bedarf dies definitiv einer Erklärung, **32** ist hier also ganz offensichtlich eine „magische“ Zahl.

10.11.2 Strukturierte Werte einlesen

Die Klasse **Reading** wird nur zum Lesen der Eingabe benötigt und ist noch einfacher aufgebaut:

```
struct Reading {
  int day;
  int hour;
  double temperature;
};

istream& operator>>(istream& is, Reading& r)
// liest die Temperaturdaten von is nach r ein
// Format: ( 3 4 9.7 )
// prüft Format, kümmert sich aber nicht um ungültige Werte
{
  char ch1;
  if (is>>ch1 && ch1!='(') {    // kann es eine Ablesung sein?
    is.unget();
    is.clear(ios_base::failbit);
    return is;
  }

  char ch2;
  int d;
  int h;
  double t;
  is >> d >> h >> t >> ch2;
  if (!is || ch2!=')') error("ungueltige Ablesung");    // Fehler beim Einlesen
  r.day = d;
  r.hour = h;
  r.temperature = t;
  return is;
}
```

Gleich als Erstes prüfen wir, ob die Eingabe im korrekten Format beginnt. Ist dies nicht der Fall, setzen wir den Dateistatus auf **fail()** und kehren zurück. Auf diese Weise halten wir uns die Option offen, das Einlesen der Informationen noch auf eine andere Weise versuchen zu können. Wenn wir erst später, nachdem wir bereits einige Daten eingelesen haben, einen Verstoß gegen das erwartete Format feststellen, ist eine Wiederaufnahme der Eingabeoperation nicht mehr realistisch und wir steigen mit **error()** aus.

Die Eingabeoperation für **Month** ist analog aufgebaut, nur dass statt einer festen Gruppe von Werten eine beliebige Zahl von Messwerten eingelesen wird:

```
istream& operator>>(istream& is, Month& m)
// lies die Daten eines Monats von is nach m ein
// Format: { month feb ... }
{
  char ch = 0;
  if (is >> ch && ch!='{') {
    is.unget();
    is.clear(ios_base::failbit); // konnten keinen Monatseintrag einlesen
    return is;
  }

  string month_marker;
  string mm;
  is >> month_marker >> mm;
  if (!is || month_marker!="month") error("falscher Beginn des Monatseintrags");
  m.month = month_to_int(mm);

  Reading r;
  int duplicates = 0;
  int invalids = 0;
  while (is >> r) {
    if (is_valid(r)) {
      if (m.day[r.day].hour[r.hour] != not_a_reading)
        ++duplicates;
      m.day[r.day].hour[r.hour] = r.temperature;
    }
    else
      ++invalids;
  }

  if (invalids) error("ungueltiger Eintrag für Monat",invalids);
  if (duplicates) error("doppelter Eintrag für Monat",duplicates);
  end_of_loop(is,'}',"falsches Ende des Monatseintrags");
  return is;
}
```

Auf die Funktion **month_to_int()** werden wir später noch eingehen; ihr obliegt die Aufgabe, die symbolische Repräsentation eines Monats (wie z.B. **jun**) in eine Zahl im Bereich [0:11] umzuwandeln. Beachten Sie die Verwendung der Funktion **end_of_loop()** aus §10.10, mit der wir das Abschlusszeichen überprüfen, und sehen Sie sich auch an, wie wir ungültige und doppelte Ablesungen registrieren, vielleicht interessiert sich ja jemand dafür.

Vor dem Speichern einer Ablesung überprüft der **>>**-Operator von **Month** kurz, ob das für die Ablesung aufgebaute **Reading**-Objekt plausible Werte enthält:

```
const int implausible_min = -200;
const int implausible_max = 200;

bool is_valid(const Reading& r)
// ein grober Test
{
  if (r.day<1 || 31<r.day) return false;
  if (r.hour<0 || 23<r.hour) return false;
  if (r.temperature<implausible_min|| implausible_max<r.temperature)
    return false;
  return true;
}
```

Schließlich müssen wir noch Jahreseinträge einlesen. Der **>>**-Operator von **Year** gleicht dem **>>**-Operator von **Month**:

```
istream& operator>>(istream& is, Year& y)
// lies die Daten eines Jahres von is nach y ein
// Format: { year 1972 ... }
{
  char ch;
  is >> ch;
  if (ch!='{') {
    is.unget();
    is.clear(ios::failbit);
    return is;
  }

  string year_marker;
  int yy;
  is >> year_marker >> yy;
  if (!is || year_marker!="year") error("falscher Beginn des Jahreseintrags");
  y.year = yy;

  while(true) {
    Month m; // in jedem Schleifendurchgang ein frisches m verwenden
    if(!(is >> m)) break;
    y.month[m.month] = m;
  }

  end_of_loop(is,'}',"falsches Ende des Jahreseintrags");
  return is;
}
```

Eigentlich wollten wir schreiben, dass sich die beiden Operatorfunktionen „bis aufs Haar“ gleichen, doch es gibt da einen entscheidenden Unterschied. Sehen Sie sich die Einleseschleife an. Hätten Sie nicht eher Code wie den folgenden erwartet?

```
Month m;
while (is >> m)
  y.month[m.month] = m;
```

Eigentlich hätten Sie genau solchen Code erwarten müssen, denn auf diese Weise haben wir bisher alle Einleseschleifen geschrieben. Auch den **>>**-Operator von **Month** haben wir zunächst nach diesem Muster geschrieben – aber es hat sich als falsch erwiesen. Das Problem ist, dass **operator>>(istream& is, Month& m)** der Variablen **m** keinen komplett neuen Wert zuweist, sondern **m** einfach die Daten der Ablesungen hinzufügt. Hoppla! Die wiederholten **is>>m**-Operationen würden also allesamt in ein und dasselbe **m** einlesen, mit dem Effekt, dass jeder neue Monat auch die Ablesungen der vorangehenden Monate dieses Jahres enthielte. Wir aber benötigen für jede **is>>m**-Operation ein neues, jungfräuliches **Month**-Objekt, in das wir einlesen können. Der einfachste Weg dies zu erreichen war, die Definition von **m** in die Schleife zu verschieben, sodass das Objekt in jedem Schleifendurchgang neu initialisiert wird. Die Alternativen wären gewesen, innerhalb von **operator>>(istream& is, Month& m)** dem Parameter **m** vor dem Einlesen ein leeres **Month**-Objekt zuzuweisen oder die Schleife wie folgt zu formulieren:

```
Month m;
while (is >> m) {
  y.month[m.month] = m;
  m = Month(); // m "erneut initialisieren"
}
```

Testen wir unseren Code:

```
// Eingabedatei öffnen:
cout << "Geben Sie bitte den Namen der Eingabedatei ein\n";
string name;
cin >> name;
ifstream ifs(name.c_str());
if (!ifs) error("Fehler beim Oeffnen der Eingabedatei ",name);

ifs.exceptions(ifs.exceptions() | ios_base::badbit);   // für bad() Ausnahme werfen

// Ausgabedatei öffnen:
cout << "Geben Sie bitte den Namen der Ausgabedatei ein\n";
cin >> name;
ofstream ofs(name.c_str());
if (!ofs) error("Fehler beim Oeffnen der Ausgabedatei ",name);

// lies eine unbestimmte Anzahl von Jahreseinträgen:
vector<Year> ys;
while(true) {
  Year y;     // jedes Mal ein neu initialisiertes Year-Objekt verwenden
  if (!(ifs>>y)) break;
  ys.push_back(y);
}
cout << "lies " << ys.size() << " Jahre mit Messdaten ein\n";

for (int i = 0; i<ys.size(); ++i) print_year(ofs,ys[i]);
```

Die Definition von **print_year()** sei dem Leser als Übung überlassen.

10.11.3 Austauschbare Darstellungen

Damit der >>-Operator von **Month** funktioniert, müssen wir einen Weg vorsehen, wie symbolische Monatsdarstellungen eingelesen werden können. (Der Symmetrie wegen werden wir auch gleich eine Möglichkeit zum Schreiben der symbolischen Monatsdarstellungen vorsehen.)

Eine einfache, aber auch sehr ermüdende Lösung hierfür wäre die Umwandlung durch eine Folge von **if**-Anweisungen:

```
if (s=="jan")
  m = 1;
else if (s=="feb")
  m = 2;
...
```

Tipp

Dies ist nicht nur nervtötend, es führt auch dazu, dass die symbolischen Namen der Monate direkt im Code stehen. Besser wäre es, die Namen irgendwo in einer Tabelle zu verwalten, sodass das Hauptprogramm nicht überarbeitet werden muss, wenn wir die symbolische Darstellung der Monate ändern. Wir haben uns daher entschlossen, die von den Eingabedateien verwendeten symbolischen Monatsdarstellungen in einem **vector<string>**-Container zu speichern und zu diesem eine passende Initialisierungs- und eine Suchfunktion anzubieten:

```
vector<string> month_input_tbl; // month_input_tbl[0]=="jan"

void init_input_tbl(vector<string>& tbl)
// initialisiert den Vektor mit den Darstellungen der Eingabe
{
  tbl.push_back("jan");
  tbl.push_back("feb");
  tbl.push_back("mar");
  tbl.push_back("apr");
  tbl.push_back("may");
  tbl.push_back("jun");
  tbl.push_back("jul");
  tbl.push_back("aug");
  tbl.push_back("sep");
  tbl.push_back("oct");
  tbl.push_back("nov");
  tbl.push_back("dec");
}

int month_to_int(string s)
// Ist s der Name eines Monats? Wenn ja, liefere den Index des Monats zurück [0:11],
// andernfalls -1
{
  for (int i=0; i<12; ++i) if (month_input_tbl[i]==s) return i;
  return -1;
}
```

Falls Sie sich fragen, ob es hierfür nicht in der C++-Standardbibliothek eine einfachere Lösung gibt: Es gibt tatsächlich eine einfachere Lösung, siehe **map<string,int>** in §21.6.1.