# 31

# Nationale Besonderheiten

Dieses Kapitel behandelt die folgenden Themen:

- Länderspezifische Zeichenformate einstellen
- Zeichensätze und -codierung
- Geld-, Datums- und Zahlenformate
- Konstruktion eines eigenen Formats

Die Klasse `locale` (Header `<locale>`) bestimmt die nationalen Besonderheiten von Zeichensätzen. Dazu gehören Sonderzeichen wie die deutschen Umlaute oder Zeichen mit Akzent wie in den Worten *señor* und *garçon*. Die Ordnung zum Vergleich von Zeichenketten wird dadurch definiert, das heißt zum Beispiel, ob ä unter a oder unter ae einsortiert wird, und das Erscheinungsbild für die Ein- und Ausgabe numerischer Größen (Dezimalpunkt oder -komma?) und Datumsangaben (31.1.2011 oder 1/31/2011). Die verschiedenen Kategorien werden in Facetten (englisch *facets*) unterteilt. Dieses Kapitel konzentriert sich auf die häufigsten Anwendungen. Es ist möglich, eigene Facetten zu schreiben, die von den vorhandenen abgeleitet sind. Weitere Details sind [ISOC++] und [KL] zu entnehmen.

# 31.1 Sprachumgebungen festlegen und ändern

Ein `locale`-Objekt wird von der Iostream-Bibliothek benutzt, damit die üblichen nationalen Gepflogenheiten bei der Ein- und Ausgabe eingehalten werden. Wenn von der deutschen Schreibweise von Zahlen auf die angloamerikanische umgeschaltet werden soll, wird das `locale`-Objekt des Ein- oder Ausgabestroms entsprechend ausgewechselt. Das geschieht mit der Funktion `imbue()`, der als Parameter ein `locale`-Objekt übergeben wird. Das englische Wort *imbue* bedeutet etwa »erfüllen (mit)« oder »inspirieren (mit)«.

```
using namespace std; // auch locale ist in std
// Beispiele zur Einstellung der Sprachumgebung
locale eineSprachumgebung("POSIX");
cin.imbue(eineSprachumgebung);

// global Deutsch als Sprachumgebung setzen, dabei vorherige
// globale Sprachumgebung merken:
locale deutsch("de_DE");
locale vorherigeSprachumgebung = locale::global(deutsch);
cout.imbue(locale("de_DE")); // Ausgabe deutsch formatieren
// ...
locale::global(vorherigeSprachumgebung); // Sprachumgebung zurücksetzen
```

Das Setzen der globalen Sprachumgebung, die normalerweise durch Abfrage der Betriebssystemumgebungsvariablen `LANG` voreingestellt wird, wirkt sich nicht auf existierende Streams wie `cin` oder `cout` aus, nur auf neu erzeugte – deswegen muss ggf. `imbue()` angewendet werden. Falls `LANG` nicht definiert ist, wird automatisch die C-Sprachumgebung gesetzt, auch `classic` genannt (siehe Beispiel unten). POSIX (= Portable Operating System Interface for uniX) ist eine Familie von Standards für Betriebssystemschnittstellen. Anstatt POSIX kann eine von vielen anderen Umgebungen gewählt werden, von denen einige hier aufgelistet sind:

| | |
|---|---|
| de_DE | = Deutsch für Deutschland (ISO 8859-1) |
| de_DE@euro | = Deutsch für Deutschland mit Euro-Zeichen (ISO 8859-15) |
| de_DE.utf8 | = Deutsch für Deutschland (Unicode UTF-8) |
| de_CH | = Deutsch für die Schweiz |
| en_GB | = Englisch für Großbritannien |
| en_CA | = Englisch für Kanada |
| en_US | = amerikanisches Englisch |
| es_SV | = Spanisch für El Salvador |
| ... | |

Auf Linux-Systemen kann das eingestellte Locale mit `locale` angezeigt werden. `locale -a` listet alle verfügbaren Locales auf.

```
cout.imbue(locale::classic());
```

setzt die Standardausgabe auf die C-Sprachumgebung zurück.

### 31.1.1 Die locale-Funktionen

- `locale()`
  Konstruktor, der eine Kopie des aktuellen globalen `locale`-Objekts (gegebenenfalls mit `global()` eingestellt, siehe oben) erzeugt.
- `explicit locale(const char* name)`
  `explicit locale(const string& name)`
  Konstruktor. Die übergebene Zeichenkette bzw. der String `name` ist zum Beispiel `"de_DE"`.
- `locale(const locale& other, const char* name, category cat)`
  `locale(const locale& other, string name, category cat)`
  Der Konstruktor kopiert `other` mit Ausnahme der Facetten, die in `cat` definiert sind. `cat` kann zum Beispiel `(monetary | numeric)` sein. Sie werden entsprechend `name` gewählt. Zum Typ `category` siehe Abschnitt 31.4.
- `locale(const locale& loc1, const locale& loc2, category cats)`
  Der Konstruktor kopiert `loc1` mit Ausnahme der Facetten, die in `loc2` definiert sind. Diese werden entsprechend `cats` gewählt.
- `template<class Facet> locale(const locale& other, Facet *f)`
  Der Konstruktor kopiert `other`. Falls `f` ungleich 0 ist, wird aber die Facette `Facet` durch `*f` definiert.
- `const locale& operator=(const locale& rechts)`
  Zuweisungsoperator; gibt `*this` zurück
- `bool operator==(const locale& other) const`
  `bool operator!=(const locale& other) const`
  Vergleichsoperatoren
- `template<class Facet> locale combine(const locale& other)`
  gibt eine Kopie von `*this` zurück, wobei aber die Facette `Facet` durch die entsprechende von `other` ersetzt wird.
- `string name() const`
  gibt den Namen des `locale`-Objekts zurück, falls definiert. Andernfalls wird »*« zurückgegeben.
- `bool operator()(const string& s1, const string& s2) const`
  gibt `s1 < s2` zurück. Damit kann leicht zum Beispiel ein Vektor `v` entsprechend den nationalen Zeichenvergleichsregeln, die in einem `locale`-Objekt `loc` festgelegt sind, sortiert werden (Beispiel siehe Seite 630).
- `static locale global(const locale& loc)`
  setzt das globale `locale`-Objekt. Der vorherige Wert wird zurückgegeben.
- `static const locale& classic()`
  gibt ein `locale`-Objekt für die C-Sprachumgebung zurück (entspricht `locale("C")`).
- `template<class Facet> const Facet& use_facet(const locale& loc)`
  gibt die Referenz der Facette des Typs `Facet` des `locale`-Objekts `loc` zurück. Falls eine Facette dieses Typs in `loc` nicht existiert, wird eine `bad_cast`-Exception ausgeworfen. Ein Beispiel für die Anwendung von `use_facet` finden Sie auf den Seiten 632 und 633.
- `template<class Facet> bool has_facet(const locale& loc)`
  gibt zurück, ob eine Facette des Typs `Facet` in `loc` existiert.

### Hinweis

Im Folgenden werden Objekte des Typs `locale` benutzt, um zum Beispiel eine deutsche Schreibweise für ein Datum einzustellen. Leider sind die Konventionen für die Namensgebung systemabhängig. Um festzustellen, welche Locales (wie de_DE und en_US) unterstützt werden, können Sie das folgende Programm *checklocale.cpp* abwandeln und nutzen.

**Listing 31.1:** Test von locales

```
// cppbuch/k31/checklocale.cpp
#include<clocale>
#include<iostream>
using namespace std;

int main() {
   const char* loc = setlocale(LC_ALL, 0);
   cout << "aktuell eingestellte Locale: " << loc << endl;

   const char* locales[] = {
      // Unix
      "de_DE",
      "en_US",

      // Alias-Namen für viele Systeme (nicht genormt), siehe
      // /usr/share/locale/locale.alias (Linux)
      // C:/MinGW/share/locale/locale.alias (MinGW/Windows)
      "german",
      "deutsch",
      "french",
      "polish",

      // Windows
      "German_Germany.1252",
      "English_United States.1252"
   };

   for(size_t i = 0; i < sizeof locales / sizeof locales[0]; ++i) {
      cout << locales[i] << " wird von diesem System ";
      loc = setlocale(LC_ALL, locales[i]);
      if(!loc) {
         cout << "NICHT unterstuetzt." << endl;
      }
      else {
         cout << "unterstuetzt. loc = " << loc << endl;
      }
   }
}
```

# 31.2 Zeichensätze und -codierung

Ein Zeichensatz ist eine Abbildung von Bitmustern auf Zeichen. Damit Sender und Empfänger sich beim elektronischen Datenaustausch verstehen, ist eine Konvention über die Bedeutung der Bitmuster unerlässlich. Die bekannteste ist ASCII (American Standard Code for Information Interchange). Die ASCII-Tabelle definiert die ersten 7 Bits eines Bytes, also 128 Zeichen (siehe Anhang A.3). Umlaute und andere europäische Sonderzeichen sind nicht enthalten, sodass nach und nach viele weitere Zeichensätze hinzukamen. Sehr bekannt ist ISO-8859-1, auch »Latin 1« genannt. Dieser Zeichensatz stimmt in den ersten 128 Bytes mit ASCII überein und definiert in den anderen 128 Bytes Umlaute wie ä, ö, ü und andere Sonderzeichen, sodass ISO-8859-1 mehr als 20 Sprachen abdeckt. ISO-8859-15, auch »Latin 9« genannt, ist eine Modifikation, die auch das Euro-Zeichen € enthält.

ISO-8859-1 hat den großen Vorteil, dass für die Darstellung eines Zeichens ein Byte ausreichend ist. Wenn allerdings noch arabische, chinesische, japanische und koreanische Zeichen gefragt sind, reicht ein Byte nicht. Aus diesem Grund wurde Unicode [Unic] entwickelt. Unicode hat den Anspruch, jedem Zeichen eine Codierung zuzuordnen. Wegen der Fülle der möglichen Zeichen müssen viele Zeichen als Multi-Byte-Sequenzen abgebildet werden. Es gibt mehrere Unicode-Schemata; das am weitesten verbreitete ist UTF-8 (8-Bit Unicode Transformation Format). Jedes UTF-8-Byte hat die Eigenschaften:

- Falls das höchste Bit eines Bytes 0 ist, handelt es sich bei dem Byte um ein ASCII-Zeichen. Die anderen 7 Bits definieren den ASCII-Wert. Die ersten 128 UTF-8-Zeichen stimmen daher mit dem ASCII überein.
- Falls das höchste Bit 1 ist, ist das Byte Teil einer Multi-Byte-Sequenz.

**Merke:**
Eine Zeichenkette kann nur in Kenntnis der verwendeten Codierung sinnvoll bearbeitet und interpretiert werden.

Das bedeutet auch, dass Tastatur, Editor und Anzeige in der Codierung übereinstimmen müssen. Sie haben sicher schon den Effekt bemerkt, dass Umlaute, die ein Programm auf der Konsole ausgibt, nicht korrekt dargestellt werden. Die Ursache ist die fehlende Übereinstimmung der Codierung. In Linux wird die Codierung des Betriebssystems durch die Variable `LANG` (für language) eingestellt. Oft ist `de_DE.UTF-8` üblich. Der `de_DE`-Anteil sorgt dabei für die hier übliche Dezimalpunkt- und Datumsdarstellung usw. Diese Aspekte werden Facetten genannt; Einzelheiten folgen.

**Tipp: Einstellung der Konsole auf UTF-8**
*Windows:* Durch Anklicken des Fenstersymbols ganz oben links im Eingabeaufforderungsfenster geht man über »Eigenschaften« zu den Schriftarten. Dort Lucida Console wählen. Dann *chcp 65001* eintippen.
*Linux (Suse 11.2):* Normalerweise ist die Konsole auf UTF-8 eingestellt. Falls nicht: im

Konsolenfenster oben anklicken: Einstellungen → aktuelles Profil verwalten → Erweitert. Unten rechts können die Schriftarten gewählt werden.

## C++-Zeichenliterale

»Normale« Zeichenliterale werden durch Hochkommata gekennzeichnet, zum Beispiel `'z'`. Anstelle des Zeichens `z` können alle anderen Zeichen des Basiszeichensatzes, der normalerweise in etwa ASCII entspricht, treten, außer dem Hochkomma selbst wegen der Begrenzerfunktion, dem Backslash und dem Zeilenendezeichen. Mit dem Backslash werden Sonderzeichen eingeleitet, wie die Tabelle 1.6 auf Seite 52 zeigt. Es gibt aber weitere Möglichkeiten, die durch ein Präfix markiert werden.

- `u'z'` : Ein vorangestelltes `u` besagt, dass das Literal den Typ `char16_t` hat. Es muss mit 16 Bits darstellbar sein, und sein Wert wird durch ISO 10646 definiert. ISO 10646 ist eine Norm für den *Universal Character Set* (UCS), der die Unterformate UCS-2 und UCS-4 hat, entsprechend einer Codierung in 2 bzw. 4 Bytes. Das Standardisierungsgremium für ISO 10646 arbeitet mit dem Unicode-Gremium zusammen, um die verschiedenen Definitionen nicht auseinanderlaufen zu lassen. So entspricht UCS-4 UTF-32.
- `U'z'` : Ein vorangestelltes `U` ist dementsprechend ein `char32_t`-Zeichen.
- `L'z'` : Ein großes `L` steht für ein `wchar_t`-Zeichen. Das »w« steht für »wide«. Dieser Typ ist für Zeichensätze gedacht, die nicht mit einem Byte pro Zeichen auskommen. Die 1-Byte-Zeichen heißen im Gegensatz dazu »narrow characters«.

## C++-Stringliterale

Den C++-Zeichenliteralen stehen entsprechende Stringliterale gegenüber. Die wichtigsten Typen sind:

```
"ASCII- oder ISO 8859-1-String", d.h. ein Byte pro Zeichen
L"String mit wchar_t-Zeichen"
u8"UTF-8 codierter String"
```

`u8` und weitere Möglichkeiten sind neu dazugekommen, werden von heutigen Compilern aber noch nicht unterstützt. Wenn das Programm im ISO-8859-1-Format abgespeichert wird, werden im folgenden Beispiel die den Umlauten entsprechenden Zahlenwerte ausgegeben:

**Listing 31.2:** ISO-8859-1 Codierung

```
// cppbuch/k31/narrow.cpp (ISO-8859-1-codiert abgespeichert)
#include<algorithm>
#include<iostream>
#include<string>
using namespace std;

int main() {
   locale loc;
   cout << "loc.name()= " << loc.name() << endl;
   locale dt("de_DE");
```

```
  cout << "dt.name()= " << dt.name() << endl;
  locale vorher = locale::global(dt); // dt für alles setzen
  string s("ÄÖÜäöüß");
  cout.imbue(dt);
  for(size_t i = 0; i < s.length(); ++i) {
    cout << "Zeichen " << i << ": " << s[i] << " "
         << ((int)s[i] +256) // Korrektur für signed char
         << endl;
  }
  for(size_t i = 0; i < s.length(); ++i) {
    s[i] = toupper(s[i], dt);          // siehe Text
  }
  cout << "\n Nach toupper: " << s << endl;
}
```

Die Konsole muss natürlich auch auf ISO-8859-1 eingestellt sein. Die Anweisung `s[i] =toupper(s[i], dt);` hätte auch durch `s[i] =toupper(s[i]);`, also Aufruf der entsprechenden C-Funktion ersetzt werden können, weil die `locale`-Einstellung vorher auf `de_DE` gesetzt worden ist. Die aktuelle Einstellung wird von `toupper(char)` berücksichtigt. Die ISO-8859-1-codierte Zeile

```
  string s("ÄÖÜäöüß");
```

könnte durch

```
  string s("\xC4\xD6\xDC\xE4\xF6\xFC\xDF");
```

ersetzt werden. Das wäre zwar von jedem Editor lesbar, egal mit welcher Codierungseinstellung. Aber portabel wäre auch das nicht, denn UTF-8-codiert sehen die Umlaute so aus:

```
  string s("\xC3\x84\xC3\x96\xC3\x9C\xC3\xA4\xC3\xB6\xC3\xBC\xC3\x9F");
```

**Merke:**
Ein Programm, das Zeichenketten mit nicht-ASCII-Zeichen enthält, ist *nicht portabel.*

Unter *portabel* wird hier verstanden, dass der Quellcode auf ein beliebiges anderes System übertragen und dort übersetzt werden kann, und dass das Programm unabhängig von der `locale`-Einstellung und der Umgebung dasselbe Ergebnis liefert. Im Folgenden sehen Sie ein Beispiel, das UTF-8-codiert ist und einen `wchar_t`-String benutzt.

**Listing 31.3:** Wide-String

```
1 // cppbuch/k31/wide.cpp (UTF-8 codiert abgespeichert)
2 #include<iostream>
3 #include<fstream>
4 #include<string>
5 #include<locale>
6 using namespace std;
7
8 int main() {
```

```
 9    locale dt("de_DE.utf-8");
10    locale::global(dt); // dt fuer alles setzen
11    wstring ws(L"ÄÖÜäöüß");
12    wcout.imbue(dt);
13    wcout << "Ausgabe mit wcout: " << ws << endl;
14    wcout << L"Länge=" << ws.length() << endl;
15    wcout << "sizeof(wchar_t): " << sizeof(wchar_t) << endl;
16    wofstream wdatei("ausgabe.txt");
17    wdatei << "Ausgabe in wofstream: " << ws << endl;
18    for(size_t i = 0; i < ws.length(); ++i) {
19       wdatei << "WZeichen " << i << ": "
20              << ws[i] << endl;
21    }
22    for(size_t i = 0; i < ws.length(); ++i) {
23       ws[i] = toupper(ws[i], dt);
24    }
25    wcout << "nach toupper(): "<< ws << endl;
26 }
```

Die Konsole muss natürlich auch auf UTF-8 eingestellt sein, um eine lesbare Anzeige zu bekommen. Bemerkungen zu diesem Beispiel:

- Bei einem auf UTF-8 eingestellten Editor wird der Quellcode in UTF-8 gespeichert. Nicht-ASCII-Zeichen werden also Multbyte-Sequenzen (Zeile 11).
- Mit `wcout`, der `cout`-Entsprechung für `wchar_t`-Zeichen, wird der `wstring ws` auf der Konsole ausgegeben (Zeile 13).
- Die interne Darstellung eines `wstring`s im ausführbaren Programm ist ein Array mit *n* `wchar_t`-Zeichen, wobei *n* die Anzahl der Zeichen (nicht der Bytes!) ist, wie sie in Zeile 14 angezeigt wird.
- `wchar_t` ist ein `int`-Typ. Seine Größe wird in Zeile 15 ausgegeben (auf meinem System 4 Byte).
- Die nachfolgende Ausgabe wird in eine Datei des Typs `wofstream` umgeleitet (Zeilen 16 bis 21).
- Die Funktion `toupper()` (Zeile 23) funktioniert zeichenweise unter Berücksichtigung der übergebenen `locale`-Einstellung.

Wie man sieht, ist die Arbeit mit Wide-Strings nicht so komfortabel wie mit »normalen« Strings. Auch wird die Umwandlung von Strings verschiedener Codierungen, wie es das Programm *iconv* (siehe Tipp) leistet, noch kaum unterstützt.

**Tipp**

Der Befehl `iconv` unter Linux (Windows: MinGW-Version von iconv) wandelt Dateiformate um. Aufruf zum Beispiel: `iconv -f ISO-8859-1 -t UTF-8 -o utf8.txt deDE.txt`

Dabei bedeuten `-f` (from) das Quellformat, `-t` (to) das Zielformat. Nach `-o` folgt der Name der Ausgabedatei; zuletzt der Name der zu konvertierenden Datei. `iconv -l` zeigt alle dem Programm bekannten Zeichensatzcodierungen an. Dabei sind einige unter mehreren Namen aufgeführt. Zum Beispiel sind die ISO-10464 UCS-Codes dasselbe wie UTF. Das

Wort »bekannt« bedeutet in diesem Zusammenhang nicht, dass von jeder Codierung zu jeder anderen konvertiert werden kann.

## 31.3 Zeichenklassifizierung und -umwandlung

Die Definition, ob zum Beispiel ein spezielles Zeichen ein Buchstabe oder etwas anderes ist, hängt von der Sprache ab. Aus diesem Grund gibt es die Funktionen der Tabellen 35.2 und 35.1 (Seite 875 f.) in einer sprachumgebungsabhängigen Variante für verschiedene Zeichentypen `charT`:

```
// Zeichenklassifizierung
template<typename charT> bool isspace (charT c, const locale& loc);
template<typename charT> bool isprint (charT c, const locale& loc);
template<typename charT> bool iscntrl (charT c, const locale& loc);
template<typename charT> bool isupper (charT c, const locale& loc);
template<typename charT> bool islower (charT c, const locale& loc);
template<typename charT> bool isalpha (charT c, const locale& loc);
template<typename charT> bool isdigit (charT c, const locale& loc);
template<typename charT> bool ispunct (charT c, const locale& loc);
template<typename charT> bool isxdigit(charT c, const locale& loc);
template<typename charT> bool isalnum (charT c, const locale& loc);
template<typename charT> bool isgraph (charT c, const locale& loc);
```

```
// Zeichenumwandlung
template<typename charT> charT toupper(charT c, const locale& loc);
template<typename charT> charT tolower(charT c, const locale& loc);
```

## 31.4 Kategorien

Locale-Sprachumgebungen enthalten verschiedene Kategorien, die in Facetten unterteilt sind. Das abfragbare Datum `locale::category` ist eine `int`-Bitmaske, die die Oder-Verknüpfung aller oder eines Teils der folgenden Konstanten ist: `none`, `ctype`, `monetary`, `numeric`, `time` und `messages`. Jede dieser Kategorien definiert eine Menge lokaler Facetten, wie die Tabelle 31.1 zeigt. Die Facetten sind Template-Klassen, die als Argument den Typ `char` oder `wchar_t` für wide characters haben können.

**Tabelle 31.1:** Kategorien und Facetten (`charT`-Template-Typ)

| Kategorie | Facetten | Zweck |
|---|---|---|
| `collate` | `collate<charT>` | Zeichenvergleich |
| `ctype` | `ctype<charT>` | Zeichenklassifizierung |
| | `codecvt<char, char, mbstate_t>` | Zeichenkonvertierung |
| `numeric` | `numpunct<charT>` | Zahlenformatierung |
| | `num_get<charT>` | Eingaben |
| | `num_put<charT>` | Ausgaben |
| `monetary` | `moneypunct<char, Intl = false>` | Währungsformatierung (`Intl = true` für internationale Festlegungen) |
| | `money_get<charT>` | Eingaben |
| | `money_put<charT>` | Ausgaben |
| `time` | `time_get<charT>` | Zeiteingaben |
| | `time_put<charT>` | Zeitausgaben |
| `messages` | `messages<charT>` | Strings aus Message-Katalogen holen |

## 31.4.1 collate

Die Klasse `template<typename charT> class collate` ist eine Facette, die die Funktionen für Vergleiche von Zeichenketten kapselt. Sie besitzt die folgenden öffentlichen Elementfunktionen:

- `int compare(const charT* low1, const charT* high1, const charT* low2,const charT* high2) const`
  Diese Funktion vergleicht zwei Zeichenketten, die durch die Intervalle `[low1, high1)` und `[low2, high2)` definiert werden (zur Definition von Intervallen siehe Seite 767). Es wird 1 zurückgegeben, falls die erste Zeichenkette größer als die zweite ist, –1 im umgekehrten Fall und 0 bei Gleichheit. Der Operator `locale::operator()()` von Seite 823 ruft diese Funktion auf. Das Beispiel auf Seite 630 zeigt, wie ein `locale`-Objekt zur sprachlich korrekten Sortierung eingesetzt wird.

Mehr zur sprachlich richtigen Sortierung lesen Sie auf Seite 629.

- `basic_string<charT> transform(const charT* low, const charT* high) const`
  gibt das String-Äquivalent des Bereichs zurück, wobei die Ordnungsrelationen erhalten bleiben, d.h. ein Vergleich zweier erzeugter Strings mit dem Algorithmus `lexicographical_compare` (siehe Seite 665) muss zum selben Ergebnis wie `compare()` führen.
- `long hash(const charT* low, const charT* high) const`
  gibt einen Hash-Wert für den übergebenen Bereich zurück. Dabei ist gewährleistet, dass der Hash-Wert auch bei unterschiedlichen Werten im Bereich stets derselbe ist, wenn nur `compare()` die Bereiche als gleich ansieht, d.h. 0 zurückgibt. Ein Beispiel dafür könnte sein, dass ä und ae bei einer Sortierung gleich behandelt werden sollen.

Ein Beispiel für die Benutzung von `collate` finden Sie in der Klasse `Stringvergleich` auf Seite 633.

### 31.4.2 ctype

Die Klasse `template<typename charT> class ctype` kapselt die Funktionen für Zeichenklassifizierung und -umwandlung. So gibt zum Beispiel der Aufruf der Funktion `toupper(c, loc)` von Seite 829 für Zeichen des Typs `char` nichts anderes als `use_facet<ctype<char> >(loc).toupper(c)` zurück. `ctype` erbt von der Basisklasse `ctype_base`, die eine Maske `mask` für Klassifizierungszwecke etwa wie folgt definiert:

```
enum mask {
   space = 1<<0, print = 1<<1, cntrl = 1<<2,
   upper = 1<<3, lower = 1<<4, alpha = 1<<5,
   digit = 1<<6, punct = 1<<7, xdigit = 1<<8,
   alnum = alpha|digit, graph = alnum|punct
};
```

Die öffentliche Schnittstelle enthält die folgenden Methoden:

- `bool is(mask m, charT c) const`
  gibt zurück, ob `c` zur Klassifizierung `m` passt.
- `const charT* is(const charT* low, const charT* high, mask* vec) const`
  Diese Funktion berechnet einen Wert `M` vom Typ `ctype_base::mask` für jedes der Zeichen im Intervall `[low, high)` und legt das Ergebnis im Array `vec` beginnend an der Stelle `vec[0]` ab. `high` wird zurückgegeben.
- `const charT* scan_is(mask m, const charT* low, const charT* high) const`
  gibt einen Zeiger auf das erste Zeichen im Intervall `[low, high)` zurück, das der Klassifizierung `m` genügt. Existiert kein solches Zeichen, wird `high` zurückgegeben.
- `const charT* scan_not(mask m, const charT* low, const charT* high) const`
  gibt einen Zeiger auf das erste Zeichen im Intervall `[low, high)` zurück, das *nicht* der Klassifizierung `m` genügt. Existiert kein solches Zeichen, wird `high` zurückgegeben.
- `charT toupper(charT c) const`
  gibt den entsprechenden Großbuchstaben zurück, sofern ein solcher existiert. Andernfalls wird das Argument zurückgegeben.
- `const charT* toupper(charT* low, const charT* high) const`
  verwandelt alle Zeichen im Bereich `[low, high)` in Großbuchstaben, sofern solche existieren. Es wird `high` zurückgegeben. Vergleichbares Beispiel siehe bei `tolower()` unten.
- `charT tolower(charT c) const`
  gibt den entsprechenden Kleinbuchstaben zurück, sofern ein solcher existiert. Andernfalls wird das Argument zurückgegeben.
- `const charT* tolower(charT* low, const charT* high) const`
  verwandelt alle Zeichen im Bereich `[low, high)` in Kleinbuchstaben, sofern solche existieren. Es wird `high` zurückgegeben. Im Beispiel von Seite 826 könnte der String `s` unter Verwendung des `locale`-Objekts `dt` wie folgt in Kleinbuchstaben umgewandelt werden:

  ```
  for(size_t i = 0; i < s.length(); ++i) {
     s[i] = std::use_facet<std::ctype<charT> >(dt).tolower(s[i]);
  }
  ```

Der Vorspann `use_facet<ctype<char> >(dt)` gibt die Facette `ctype` des `locale`-Objekts `dt` zurück, deren Funktion `tolower()` dann aufgerufen wird.

- `charT widen(char c) const`
  wandelt `c` in eine entsprechende Repräsentation des Typs `charT` um (z.B. wide character `wchar_t`).
- `const char* widen(const char* low, const char* high, charT* to) const`
  wandelt jedes Zeichen im Intervall `[low, high)` in eine entsprechende Repräsentation des Typs `charT` um und legt das Ergebnis in `to` ab. Der Rückgabewert ist `high`.
- `char narrow(charT c, char dfault) const`
  wandelt `c` in eine entsprechende Repräsentation des Typs `char` um, falls eine solche existiert. Andernfalls wird `dfault` zurückgegeben.
- `const charT* narrow(const charT* low, const charT*, char vorgabe, char* to) const`
  wandelt jedes Zeichen im Intervall `[low, high)` in eine entsprechende Repräsentation des Typs `char` um, falls eine solche existiert. Andernfalls wird `vorgabe` genommen. Das Ergebnis wird in `to` abgelegt. Der Rückgabewert ist `high`.

Es existiert eine Spezialisierung `ctype<char>`.

### codecvt-Zeichensatzkonvertierung

Die Klasse `template<class internT, class externT, class stateT> class codecvt` dient zur Konvertierung von Zeichensätzen, zum Beispiel von Multibyte-Zeichen nach Unicode. Standardmäßig ist die Implementierung `codecvt<wchar_t, char, mbstate_t>` zur Konvertierung zwischen dem `char`-Zeichensatz und dem Zeichensatz für Wide Characters vorgesehen. Die Interna der Template-Klasse `stateT` bzw. `mbstate_t` sind dem jeweiligen Hersteller der C++-Standardbibliothek vorbehalten. Einzelheiten siehe [ISOC++].

## 31.4.3 numeric

Die Template-Klassen `num_get` und `num_put` wickeln das formatierte Einlesen bzw. die formatierte Ausgabe ab. Sie werden intern von den Standard-Iostreams benutzt, um Zahlen mit national bedingten Dezimal- und Tausendermarkierungen richtig zu bearbeiten, und sind für normale Benutzer/innen wohl kaum von Bedeutung, da sie versteckt innerhalb des <<- bzw. >>-Operators Anwendung finden, wie das folgende Beispiel zeigt:

```
// cppbuch/k31/inout.cpp
#include <iostream>
#include <locale>
using namespace std;

int main() {
  cin.imbue(locale("de_DE"));
  cout.imbue(locale("en_US"));
  double f;
  while (cin >> f)      // implizite Nutzung von num_get
    cout << f << endl;  // implizite Nutzung von num_put
}
```

Mit den gegebenen `locale`-Objekten würde die Eingabe 3.456,78 die Ausgabe 3,456.78 bewirken. Die Abfrage der Markierungen und anderer Dinge mit der Klasse `numpunct` folgt:

### numpunct

Die Facette `template<class charT> class numpunct` hat die folgende öffentliche Schnittstelle:

- `charT decimal_point() const`
  gibt den verwendeten Dezimalpunkt zurück (z. B. einen Punkt für en_US oder ein Komma für de_DE).
- `charT thousands_sep() const`
  gibt das Trennzeichen zwischen Tausender-Gruppen zurück.
- `string grouping() const`
  Die Zeichen des zurückgegebenen Strings, im Folgenden `S` genannt, sind als *ganzzahlige* Zahlen zu interpretieren, die die Anzahl der Ziffern in der Gruppe angeben, beginnend mit Position 0 als am weitesten rechts stehende Gruppe. Wenn `S.size() <= i` für eine Position `i` gilt, ist die Zahl dieselbe wie die für Position `i-1`. Zum Beispiel wird die Anzahl der Ziffern einer Tausendergruppe gleich 3 sein, d.h. `S == "\003"`. Negative Zahlen charakterisieren unbegrenzte Gruppen wie etwa Zahlen ganz ohne Markierung der Tausender.
- `basic_string<charT> truename() const`
  `basic_string<charT> falsename() const`
  geben den verwendeten Namen (`true` bzw. `false`) für die Ausgabe zurück, sofern `boolalpha == true` ist (vgl. Abschnitt 10.1.1, Seite 378).

Die Klasse kann für ein bestimmtes `locale`-Objekt wie folgt benutzt werden:

```
locale loc;  // Kopie des aktuellen globalen locale-Objekts
char dezPunkt = use_facet<numpunct<char> >(loc).decimal_point();
// oder
string wahr = use_facet<numpunct<char> >(loc).truename();
cout << wahr;              // Ausgabe: true
```

## 31.4.4 monetary

Diese Kategorie enthält alles, was für die formatierte Ein- und Ausgabe von Geldbeträgen einschließlich der Währungsangaben gebraucht wird. In den folgenden Beispielen wird von einer einfachen Klasse `Geld` ausgegangen.

**Listing 31.4:** Klasse Geld

```
// cppbuch/k31/geld/Geld.h
#include<iostream>

class Geld {
 public:
  Geld(long int b = 0L);
  long int getBetrag() const;
 private:
   long int betrag;
};

std::istream& operator>>(std::istream& is, Geld& G);
std::ostream& operator<<(std::ostream& os, const Geld& G);
```

### moneypunct

Die Facette `template<class charT> class moneypunct` definiert Währungssymbole und die Formatierung. Sie erbt von der Klasse `money_base`, die die öffentlichen Elemente

```
enum part { none, space, symbol, sign, value};
struct pattern { char field[4]};
```

bereitstellt. Ein monetäres Format wird durch eine Folge von vier Komponenten spezifiziert, die in einem `pattern`-Objekt `p` zusammengefasst werden. Das Element `static_cast<part>(p.field[i])` bestimmt die i-te Komponente des Formats. Aus Effizienzgründen ist `field` vom Typ `char` anstatt vom Typ `part`. Im Feld eines `pattern`-Objekts kann eines der Elemente von `part` genau einmal vorkommen. Die Klasse `money_punct` hat die folgende öffentliche Schnittstelle:

- `charT decimal_point() const`
  gibt den verwendeten Dezimalpunkt zurück.
- `charT thousands_sep() const`
  gibt das Trennzeichen zwischen Tausender-Gruppen zurück.
- `string grouping() const`
  Die Funktion hat dieselbe Bedeutung wie die gleichnamige Funktion der Klasse `numpunct` (Seite 833).
- `basic_string<charT> curr_symbol() const`
  gibt das Währungssymbol zurück, z.B. $. Für internationale Instanziierungen (vgl. Tabelle 31.1, Seite 830) werden im Allgemeinen drei Buchstaben und ein Leerzeichen zurückgegeben, z.B. »`USD` «.
- `basic_string<charT> positive_sign() const` und
  `basic_string<charT> negative_sign() const`
  geben das Zeichen für einen positiven Wert ('+' oder Leerzeichen) bzw. einen negativen Wert (meistens '-') zurück.
- `int frac_digits() const`
  gibt die Ziffern nach dem Dezimalpunkt an, im Allgemeinen zwei.
- `pattern pos_format() const` und
  `pattern pos_format() const`
  geben das benutzte Formatierungsmuster zurück. Das Standardmuster ist `{symbol, sign, none, value}`.

Die Klasse kann für ein bestimmtes `locale`-Objekt wie folgt benutzt werden:

```
locale loc;  // Kopie des aktuellen globalen locale-Objekts
char dezPunkt = use_facet<moneypunct<char> >(loc).decimal_point();
```

### money_get

Die Template-Klasse `template<class charT> class money_get` wickelt das formatierte Einlesen von Geldbeträgen, ggf. mit Währungsangaben, ab. Sie hat zwei öffentliche Methoden

- ```
  iter_type get(iter_type s, iter_type end, bool intl,
                ios_base& f, ios_base::iostate& err,
                long double& units) const
  ```
  ```

- `iter_type get(iter_type s, iter_type end, bool intl,`
  `ios_base& f, ios_base::iostate& err,`
  `string_type& units) const`

`iter_type` ist eine öffentliche, in der Klasse definierte Typbezeichnung für einen Input-Iterator, dessen Typ mit `istreambuf_iterator<charT>` vorgegeben ist. Dieser Typ wird nicht weiter beschrieben, weil erstens der Typ über den Namen `money_get::iter_type` benutzbar ist und er zweitens im Allgemeinen nicht alleinstehend benötigt wird, wie das Beispiel unten zeigt. `string_type` ist der in der Klasse definierte Name für den Typ `basic_string<charT>`. Diese Methoden lesen einen Geldbetrag als `double`-Zahl bzw. einen String ein, wobei der Dezimalpunkt eliminiert wird. Sie können in einer benutzerdefinierten Klasse zur Implementierung des Eingabeoperators (`>>`) verwendet werden. Zurückgegeben wird ein Iterator, der auf das unmittelbar nach dem letzten gültigen Zeichen eines Geldbetrags folgende Zeichen verweist. Im folgenden Beispiel wird Bezug auf die oben erwähnte Klasse `Geld` genommen.

**Listing 31.5:** Locale-abhängiger Eingabeoperator der Klasse Geld

```
// Auszug aus der Datei cppbuch/k31/geld/Geld.cpp
#include"Geld.h"
#include<locale>
Geld::Geld(long int b)
   :betrag(b) {
}
long int Geld::getBetrag() const {
   return betrag;
}

std::istream& operator>>(std::istream& is, Geld& geld) {
   std::istream::sentry s(is); // sentry siehe Seite 396
   if(s) {
      std::ios_base::iostate fehler = is.rdstate();
      is.setf(std::ios::showbase); // damit die Währung ausgewertet wird
      long double wieviel = 0;
      std::use_facet<std::money_get<char> >(is.getloc())
         .get(is, 0, false, is, fehler, wieviel);
      is.setstate(fehler);
      if(!fehler) {
         geld = Geld(static_cast<long int>(wieviel));
      }
      else {
         std::cerr << "fehlerhafte Eingabe!\n";
      }
   }
   return is;
}
```

Zwar ist die Basis der Klasse `Geld` ein ganzzahliger Cent-Betrag, die obige `put()`-Funktion verlangt jedoch `long double`. Aus diesem Grund wird die Typumwandlung `static_cast` eingesetzt. Das Beispiel zeigt, dass der Istream `is` an die Stelle des verlangten Input-Iterators treten kann. Der Grund liegt darin, dass die Klasse `istreambuf_iterator<charT>` einen Konstruktor hat, der ein `istream`-Objekt als Parameter nimmt. Der vierte Parameter

von `get()` nutzt aus, dass die Klasse `istream` von der Klasse `ios_base` erbt. Er dient dazu, intern über `getloc()` auf die Facette `moneypunct` zuzugreifen. Das folgende Programmfragment zeigt eine Anwendung:

```
Geld dollars;
cin.imbue(locale("en_US"));
cin >> dollars;
```

Eine Zeichenfolge `"1056.23"` im Eingabestrom führt zu dem Ergebnis

```
dollars.betrag == 105623.
```

### money_put

Die Template-Klasse `template<class charT> class money_put` wickelt die formatierte Ausgabe von Geldbeträgen, gegebenenfalls mit Währungsangaben, ab. Sie hat zwei öffentliche Methoden:

- `iter_type put(iter_type s, bool intl, ios_base& f, charT fill, long double& units) const`
- `iter_type put(iter_type s, bool intl, ios_base& f, charT fill, string_type& digits) const`

Die Typbezeichnungen entsprechen denen der Klasse `money_get`, wobei der Typ `iter_type` natürlich ein Output-Iterator ist. Anwendungsmöglichkeiten ergeben sich in Analogie zur Klasse `money_get`, zum Beispiel der Ausgabeoperator für die obige Klasse `Geld`:

**Listing 31.6:** Locale-abhängiger Ausgabeoperator der Klasse Geld

```
// Auszug aus der Datei cppbuch/k31/geld/Geld.cpp
// .. Fortsetzung von oben

std::ostream& operator<<(std::ostream& os, const Geld& geld) {
   std::ostream::sentry s(os);
   os.setf(std::ios::showbase); // damit die Währung angezeigt wird
   if(s) {
      std::use_facet<std::money_put<char> >(os.getloc())
         .put(os, true, os, ' ', static_cast<double>(geld.getBetrag()));
   }
   return os;
}
```

## 31.4.5 time

Diese Kategorie enthält zwei Klassen, die für die formatierte Ein- und Ausgabe von Zeiten und Datumsangaben gebraucht werden können.

### time_get

Die Template-Klasse `template<class charT> class time_get` wickelt das formatierte Einlesen von Datumsangaben und Zeiten ab. Die Klasse erbt von der Klasse `time_base`, die den öffentlichen Typ `dateorder` zur Verfügung stellt:

```
enum dateorder { no_order, dmy, mdy, ymd, ydm}
```

Dieser Typ spezifiziert die möglichen Ordnungen (dmy = day month year usw.). Die Klasse `time_get` deklariert den Typ `iter_type` für einen Input-Iterator, dessen Typ mit `istreambuf_iterator<charT>` vorgegeben ist. Wie bei der Klasse `money_get` ist auch hier die genaue Typkenntnis nicht notwendig (siehe Beispiel unten). Die öffentlichen Methoden der Klasse `time_get` sind:

- `dateorder date_order() const`
  liefert die verwendete Reihenfolge von Tag, Monat und Jahr.
- `iter_type get_time(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t) const`
- `iter_type get_date(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t) const`
- `iter_type get_weekday(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t) const`
- `iter_type get_monthname(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t) const`
- `iter_type get_year(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err, tm* t) const`

Alle `get`-Methoden ermitteln über den Parameter `f` die verwendete Sprachumgebung und das Format. Der Typ `tm` ist auf Seite 881 beschrieben. Die Methoden lesen ab Position `s` alle Zeichen, die notwendig sind, die Struktur `*t` bezüglich der gewünschten Information (Zeit, Datum, Wochentag, Monatsname, Jahr) zu füllen bzw. bis ein Fehler auftritt. Zurückgegeben wird ein Iterator auf die Position direkt nach dem letzten Zeichen, das noch zu der gelesenen Information gehört. Das Beispiel zeigt, wie der Eingabeoperator für die Klasse `Datum` aus Abschnitt 9.3, Seite 334, realisiert werden kann.

**Listing 31.7:** Locale-abhängiger Eingabe-Operator

```
// Auszug aus der Datei cppbuch/k31/datum/datum.cpp
#include"datum.h"       // Deklarationen und #include<iostream> dort nachtragen

std::istream& operator>>(std::istream& is, Datum& d) {
   std::istream::sentry s(is);
   if(s) {
      std::ios_base::iostate fehler = std::ios_base::goodbit;
      struct std::tm t;
      std::use_facet<std::time_get<char> >(is.getloc())
         .get_date(is, 0, is, fehler, &t);
      if (!fehler) {
         d = Datum(t.tm_mday, t.tm_mon + 1, t.tm_year + 1900);
      }
      is.setstate(fehler);
   }
   return is;
}
```

### time_put

Die Template-Klasse `template<class charT> class time_put` wickelt die formatierte Ausgabe von Datumsangaben und Zeiten ab. Die Klasse hat den öffentlichen Typ `iter_type` für einen Output-Iterator und die folgenden öffentlichen Methoden:

- `iter_type put(iter_type s, ios_base& f, charT fill, const tm* tmb, const charT* pat, const charT* pat_end) const`
  Diese Methode gibt die in der Struktur `tmb` liegende Zeit entsprechend einem Muster aus, das im Formatstring von `pat` bis `pat_end` vorliegt. Das Muster entspricht den für `strftime()` (Seite 882) üblichen Konventionen. `fill` ist ein Füllzeichen, zum Beispiel das Leerzeichen.
- `iter_type put(iter_type s, ios_base& f, charT fill, const tm* tmb, char format, char modifier = 0) const;`
  Diese Methode gibt die in der Struktur `tmb` liegende Zeit entsprechend einem Muster aus, das im Zeichen `format` definiert ist. Dieses Zeichen ist eins der möglichen, die nach dem '%'-Zeichen im `strftime()`-Format vorkommen können. Der Parameter `modifier` ist implementationsabhängig.

Das folgende Beispiel zeigt, wie der Ausgabeoperator (vgl. Aufgabe auf Seite 337) und die Methode `toString()` (vgl. Aufgabe auf Seite 339) realisiert werden können.

**Listing 31.8:** Locale-abhängiger Ausgabe-Operator und toString()

```
// Auszug aus der Datei cppbuch/k31/datum/datum.cpp
std::ostream& operator<<(std::ostream& os, const Datum& d) {
   std::ostream::sentry s(os);
   if(s) {
      struct std::tm t;
      t.tm_mday = d.tag();
      t.tm_mon  = d.monat()-1;
      t.tm_year = d.jahr()-1900;
      std::use_facet<std::time_put<char> >(os.getloc())
         .put(os, os, ' ', &t, 'x'); // x : siehe strftime
   }
   return os;
}

std::string Datum::toString(const std::locale& loc) const {
   std::ostringstream oss; // siehe Seite 393
   oss.imbue(loc);
   oss << *this;     // Benutzung des obigen operator<<()
   return oss.str();
}
```

Eine kleines Beispiel demonstriert die Anwendung von `time_get` und `time_put` mit den neu definierten Ein- und Ausgabeoperatoren:

**Listing 31.9:** Datumsformate

```
// cppbuch/k31/datum/main.cpp
#include"datum.h"
using namespace std;
```

```
int main() {
  Datum einDatum;
  locale deDE("de_DE");
  cout << " bitte Datum im Format tt.mm.yyyy eingeben:" << endl;
  cin.imbue(deDE);
  cin >> einDatum;
  cout.imbue(deDE);
  cout << "deutsches Format :" << einDatum << endl;
  locale enUS("en_US");
  cout.imbue(enUS);
  cout << "US Format      :" << einDatum << endl;
  cout << "toString() mit Standard-Locale deDE:"
       << einDatum.toString() << endl;
  cout << "toString() mit Locale enUS        :"
       << einDatum.toString(enUS) << endl;
}
```

### 31.4.6 messages

Die Klasse `template<class charT> class messages` implementiert das Holen von Meldungen aus Katalogen. Wie ein Katalog realisiert ist, ob zum Beispiel als Datei oder Teil einer Datenbank, ist implementationsabhängig. Der Typ `catalog`, ein `int`-Typ, steht für eine Katalognummer. Es gibt die folgenden Elementfunktionen:

- `catalog open(const string& fn, const locale&) const`
  eröffnet den Katalog, der durch den String `fn` identifiziert wird, und gibt eine Identifizierungszahl zurück, die bis zum folgenden `close()` zu dem Katalog gehört. Falls diese Zahl negativ ist, kann der Katalog nicht geöffnet werden.
- `void close(catalog c)`
  schließt den Katalog `c`.
- `basic_string<charT> get(catalog c, int set, int msgid,`
  `const basic_string<charT>& vorgabe) const`
  Es wird die durch die Argumente `set`, `msgid` und `vorgabe` identifizierte Meldung zurückgegeben. Falls keine Meldung gefunden wird, ist `vorgabe` das Ergebnis.

## 31.5 Konstruktion eigener Facetten

Man kann vorhandene Facetten durch eigene ersetzen. Dazu muss man wissen, dass zu allen Methoden der oben beschriebenen Facetten zusätzliche virtuelle Methoden mit exakt denselben Schnittstellen existieren, die ein vorangestelltes `do_` im Namen haben und `protected` sind. Diese Methoden werden von den oben beschriebenen aufgerufen. Ferner gibt es Klassen, die von den beschriebenen Facetten nur die `protected`-Schnittstelle erben und im Namen ein nachgestelltes `_byname` tragen, um auszudrücken, dass Namen für die Facetten vergeben werden können. Von diesen Klassen können eigene Klassen abgeleitet und die Methoden überschrieben werden. Das folgende Beispiel zeigt, wie das

vorgegebene Standardsymbol für Euro, nämlich EUR, mithilfe einer eigenen Facette für Währungssymbole durch das Symbol € ersetzt wird.

```
// cppbuch/k31/euro.cpp
#include <iostream>
#include <locale>
#include <string>
#include"Geld.h"
typedef std::moneypunct_byname<char,true> MeinMoneypunct;

class MeinWaehrungsformat : public MeinMoneypunct {
 protected:
   // Redefinieren der virtuellen Funktion do_curr_symbol(), die von der
   // public-Funktion curr_symbol() der Basisklasse moneypunct gerufen wird:
   std::string do_curr_symbol() const {
      return wsymbol;
   }
 public:
   MeinWaehrungsformat(const std::locale& loc, const char* ws)
      : MeinMoneypunct(loc.name().c_str()), wsymbol(ws) {
   }
 private:
   const char* wsymbol;
};

using namespace std;

int main() {
   locale locUS("en_US");
   Geld derBetrag;
   cout << "Eingabe in Cent(!), z.B. 123456:?";
   cin >> derBetrag;
   cout << "direkte Abfrage mit voreingestellter locale ("
        << locale().name() << ") :"                          // locale 'C'
        << derBetrag.getBetrag() << endl;                    // 123456
   cout.imbue(locUS);       // cout auf enUS umschalten
   cout << "Es wurde " << derBetrag
        << " eingegeben (US-Format).\n";                     // USD 1,234.56
   locale deDEeuro("de_DE@euro");
   cout  << "Ausgabe Standard-Währungssymbol EUR und Dezimalkomma "
            "statt -punkt :" << endl;

   // Achtung: KEINE Währungsumrechnung!
   cout.imbue(deDEeuro);      // cout auf deDE@euro umschalten
   cout << derBetrag << endl;                          // 1.234,56 EUR
   MeinWaehrungsformat* mwptr = new MeinWaehrungsformat(deDEeuro,
                                "\u20ac");  // Euro-Symbol in UTF-8
                                            // \xA4 in ISO-8859-15
   cout << "Ausgabe eigenes Währungssymbol und Dezimalkomma"
           " statt Dezimalpunkt :" << endl;
   cout.imbue(locale(deDEeuro, mwptr));
   cout << derBetrag << endl;                          // 1.234,56 €
}
```