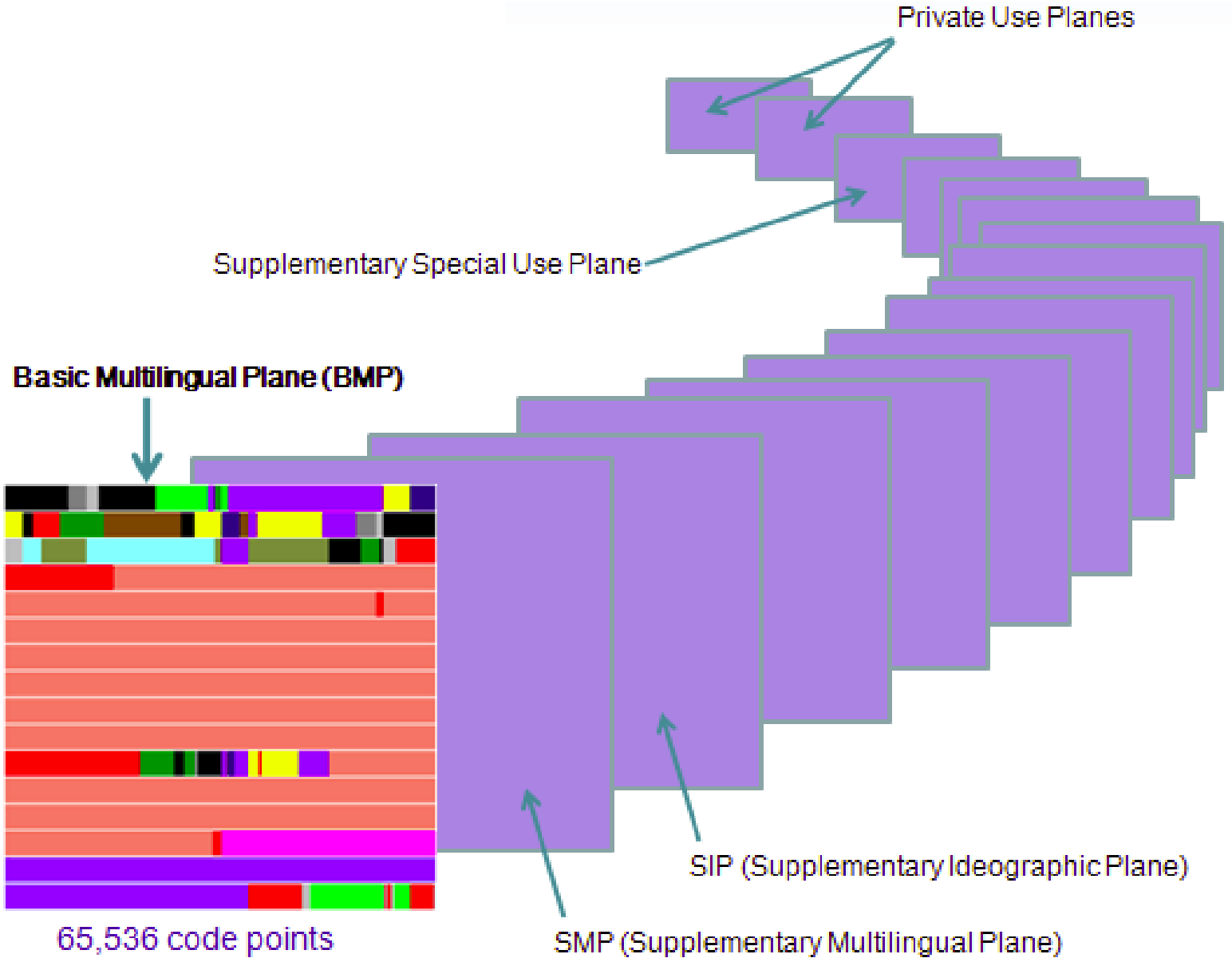
Unicode und Localization

Tilman Pannhorst

Grundlagen Unicode

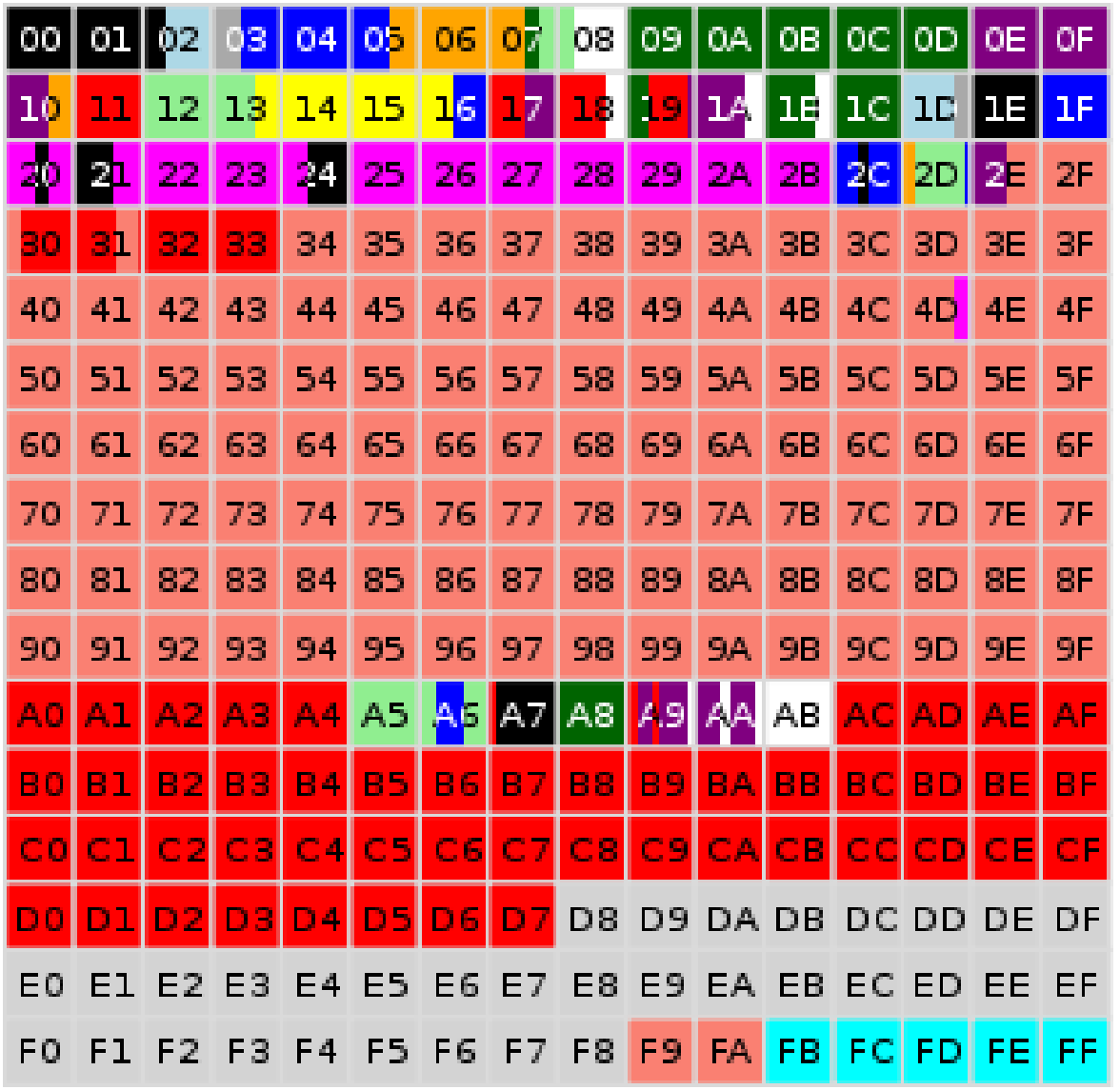
- Versuch, alle (benutzten) Schriftzeichen der Welt abzubilden
- Unicode Zeichen -> Codepoint
- Abwärtskompatibel zu ISO-8859-1 (Lateinisches Alphabet und …
- Multibyte-Encoding
- 17 Planes mit jeweils 2^16 = 65.536 Codepoints
- Erste Version 1991, aktuell Version 6.1

Unicode Planes



Basic Multilingual Plane

Basic Multilingual Plane



UTF-Codierung

- UTF: UCS Transformation Format
- UCS: Universal Character Set
  - Internationale Norm, die dem Unicode entspricht
- Gebräuchliche Kodierungen:
  - UTF-8, UTF-16, (UTF-32 / UCS 4)

UTF-8

- Codiert alle Codepoints in 8-Bit-Werten (char)
- Bestimmte chars zeigen an, dass der nächste Char auch noch dasselbe Zeichen kodiert

Unicode-Bereich (hexadezimal)	UTF-8-Kodierung (binär)	Bemerkungen	Möglichkeiten (theoretisch)	
0000 0000 – 0000 007F	0xxxxxxx	In diesem Bereich (128 Zeichen) entspricht UTF-8 genau dem ASCII-Code: Das höchste Bit ist *0*, die restliche 7-Bit-Kombination ist das ASCII-Zeichen.	2^7	128
0000 0080 – 0000 07FF	110xxxxx 10xxxxxx	Das erste Byte enthält binär 11xxxxxx, die folgenden Bytes 10xxxxxx; die *x* stehen für die fortlaufende Bitkombination des Unicode-Zeichens. Die Anzahl der Einsen vor der höchsten 0 im ersten Byte ist die Anzahl der Bytes für das Zeichen. (In Klammern jeweils die theoretisch maximal möglichen.)	$2^{11} - 2^7$ (2^{11})	1920 (2048)
0000 0800 – 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx		$2^{16} - 2^{11}$ (2^{16})	63.488 (65.536)
0001 0000 – 0010 FFFF [0001 0000 – 001F FFFF]	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx		2^{20} (2^{21})	1.048.576 (2.097.152)

UTF-8

- Abwärtskompabilität:
  - UTF-8 kodierte Texte können als Latin1 (ISO-8859-1), Windows-125x und ASCII geöffnet werden, solange sie nur “einfache” lateinische Buchstaben, Zahlen und ausgewählte Sonderzeichen enthalten
  - Probleme bereits bei Umlauten

    “ Die Selbsthilfegruppe "UTF-8-Probleme" trifft sich diesmal abweichend im grÃ¼nen Saal. ”

UTF-16

- UTF-16
  - Codiert alle Codepoints in 16Bit-Werten
  - Wichtig:
    - Für alle Codepoints außerhalb der BMP werden weiterhin mehrere 16-Bit-Werte benötigt
    - In vielen Programmen fehlerhaft umgesetzt, BSP: 𝄞
- UCS2
  - Jedes sichtbare Zeichen wird nur in einem 16-Bit-Wert gespeichert
    - Nur die BMP kann angezeigt werden

UTF32 / UCS 4

- UTF32
  - Codiert alle Codepoints in 32Bit-Werten
  - Jeder Wert ist genau ein Zeichen
    - Leichte Handhabung, aber bei lat. Buchstaben vierfacher Platzverbrauch
- UCS 4
  - Entspricht UTF32

Unicode in pre11-C++

- Vor C++11:
  - Zeichentyp *wchar_t*
    - Keine festgelegte Länge
      - Linux: 4 Byte (!)
        - UTF-32
      - Windows: 2 Byte
        - Mindestens UCS2, heute UTF-16
  - Stringtyp std::*wstring*
  - Platformen-Präferenzen
    - Linux
      - *char* verbreitet, Nutzung von UTF-8 in *std::string*
    - Windows:
      - *wchar_t* verbreitet, Funktionen der Windows API mit W-Suffix

Unicode in pre11-C++ - Beispiel

```
const char *text = "快樂";
const wchar_t *wtext = L"快樂"; // L markiert ein wide-Literal

std::string stext(text);
std::wstring swtext(wtext);


std::cout << "char: " << text << " mit Länge " << stext.size()
<< std::endl;

// funktioniert nicht, es werden nur Adressen ausgegeben (Länge
klappt)
std::cout << "wchat_t: " << wtext.c_str() << " mit Länge " <<
swtext.size() << std::endl;

// scheitert eventuell an Ausgabe, z.b. in Linux an UTF-8-
Konsole
std::wcout << "und wide: " << swtext << std::endl;
```

Unicode in C++11

- Neue Zeichentypen mit fixer Breite:
  - *char16_t*
    - 2 Byte – 16 Bit
  - *char32_t*
    - *4 Byte – 32 Bit*

Unicode in C++ 11

- Dazu neue Stringtypen
  - u16string
  - u32string

- Neue Präfixe für String-Literale:
  - u8”literal” - utf8
  - u”literal” - utf16
  - U”literal” - utf32

Unicode in C++11

- Escape-Sequenzen für einzelne Unicodezeichen
  - \u*hhhh*
    - Kann nur die BMP (16 Bit) ausdrücken
  - \U*hhhhhhhh*
    - Kann alle Codepoints annehmen (32 Bit)
  - Beide natürlich auch in UTF-8 bzw UTF-16 codierten Strings nutzbar
- Weiterhin gilt: Ein Zeichen (Codepoint) kann mehr Platz als ein *char* bzw *char16_t* brauchen
- Erst ein *char32_t* kann nach aktuellem Standard alle Unicode-Zeichen in sich aufnehmen

Unicode in C++11

- Konvertierungen mit dem codecvt-Template
  - `template <class internT, class externT, class stateT> codecvt;`
- Standard codecvt-Facetten (Auswahl):
  - `codecvt_utf8_utf16` (Zwischen UTF16 und UTF8)
  - `codecvt<char32_t, char, mbstate_t>` (Zwischen UTF32 und UTF8)
  - `codecvt_utf8` (Zwischen UTF8 und UCS2/4)
  - `codecvt_utf16` (Zwischen UTF16 und UCS2/4)
- Neu: Nutzung von wstring_convert
- `template< class Codecvt, class Elem = wchar_t, class Wide_alloc = std::allocator<Elem>, class Byte_alloc = std::allocator<char> > class wstring_convert;`

Unicode in C++11

Character conversions	narrow multibyte (char)	UTF-8 (char)	UTF-16 (char16_t)
UTF-16	mbrtoc16 / c16rtomb	codecvt<char16_t, char, mbstate_t> codecvt_utf8_utf16<char16_t> codecvt_utf8_utf16<char32_t> codecvt_utf8_utf16<wchar_t>	N/A
UCS2	No	**codecvt_utf8**<char16_t>	codecvt_utf16<char16_t>
UTF-32/UCS4 (char32_t)	mbrtoc32 / c32rtomb	codecvt<char32_t, char, mbstate_t> **codecvt_utf8**<char32_t>	codecvt_utf16<char32_t>
UCS2/UCS4 (wchar_t)	No	**codecvt_utf8**<wchar_t>	codecvt_utf16<wchar_t>
wide (wchar_t)	codecvt<wchar_t, char, mbstate_t> mbsrtowcs / wcsrtombs	No	No

Unicode in C++11 - Beispiel

```
using namespace std;

const char *text = u8"快樂";

char16_t invalid = u'𝄞';  // gibt Warning, da Zeichen außerhalb
der BMP nicht in char16_t passt
char32_t valid = U'𝄞';

u16string s1 = u"Ein UTF-16 kodierter String";
u32string s2 = U"Ein UTF-32 kodierter String";

wcout << s1.c_str() << s2.c_str() << endl; // not working
correctly on UTF8-Linux

const char16_t *c1 = s1.c_str();

// konvertiere utf16 zu utf8

wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> conv;
string s = conv.to_bytes(c1);
cout << s << endl;
```

Localization

Localization Grundlagen

- Begriffe:
  - **Internationalization**
    - Einbauen der Möglichkeit, die Software an lokale Gegebenheiten (Sprache, Währungen, Datumsformate...) anzupassen
  - **Localization**
    - Anpassen der Software durch die in der Internationalisierung gegebenen Möglichkeiten

Locales in C++

- *facet* (Facette)
  - Beinhalten Daten und Funktionen für jeweils eine der Lokalisierungen
- *locale*
  - Bündellung mehrere Facetten in einer Sammlung
  - Dürfen nach Initialisierung nicht verändert werden
  - Enthalten immer nur eine Facette aus einer Facettenfamilie
- Standard-Facettenfamilien: *ctype (character type), collate* (Sortierung von Strings)*, message, numeric, monetary, time*

Locales in C++

- Erzeugt durch Konstruktur
  - `explicit locale (const char* std_name);`
- Leider nicht platformunabhängig, da Namen nicht festgelegt:
  - Deutsche locale:
    - `locale("De_DE")` in der Glibc
    - `locale("German_Germany.1252")` unter Windows
- Zwei Festlegungen:
  - `locale("C")` oder `locale::classic()` erzeugen "U.S. English ASCII" locale
  - `locale("")` erzeugt die als Systemstandard festgelegte locale

Locales in C++ - Beispiel

```
using namespace std;

locale sys("");  // System-Locale (meistverwendet)
locale deflt("C");  // U.S. English ASCII

locale ger("de_DE.UTF-8");
locale zh("zh_CN.UTF-8");  // Simplified Chinese


locale::global(sys); // Setze programmweite locale

locale glob; // Hole programmweite locale

// gebe Namen aus
cout << "system locale: " << sys.name() << endl;
cout << "default locale: " << deflt.name() << endl;
cout << "german locale: " << ger.name() << endl;
cout << "simplified chinese: " << zh.name() << endl;
cout << "program global locale: " << glob.name() << endl;
```

Locales in C++

- Zur Nutzung müssen Facetten aus der locale geholt werden
  - `template <class Facet> bool has_facet(const locale&) throw()`
  - `template <class Facet> const Facet& use_facet(const locale&)`
    - Hinweis: Zurückgegebene Facette ist nur solange gültig wie das local-Object

Standardfacetten - *ctype*

- Klassifikation
  - `bool is(mask m, charT c) const;`
    - Wobei mask:
      - `ctype_base::lower, ::digit, ::print(able), ::space, ::alpha(betical)`
    - Alternativ: isupper, isprint, isspace etc..
    - Überladen auch für Sequenzen von Characters [beg, end]
- Konvertierung zwischen den Klassen
  - `template <class charT> charT tolower (charT c, const locale& loc) const;`
  - `template <class charT> charT toupper (charT c, const locale& loc) const;`
- Konvertierung von charT nach char

Standardfacetten – *string collation*

- Sortierung von Zeichenketten
- `template <class charT> class collate`
- `int compare(const charT* beg1, const charT*end1, const charT* beg2, const charT* end2) const;`
  - Vergleicht zwei Character-Sequenzen (nicht nullterminiert)
  - Rückgabewerte: 1 – erste Sequenz größer, 0 - beide gleich, -1 – zweite Sequenz größer
- `long hash(const charT* beg, const charT* end) const;`

Standardfacetten – *message catalog*

- Abrufen von Texten in verschiedenen Sprachen
- Speicherform und Erstellung der Kataloge implementationsabhängig
- `Template <class charT> class messages`
- Memberfunktionen:
  - `catalog open(const basic_string<char>& name, const locale& l) const;`
  - `basic_string<charT> get(catalog cat_id, int set_id, int msg_id, const basic_string<charT> & dfault) const;`
  - `void close(catalog c) const;`

Standardfacetten - *numpunct*

- `decimal_point()` - Zeichen für Radixtrennung (dt. Komma)
- `thousands_sep()` - Zeichen für Trennung von Tausendergruppen (dt. Punkt)
- `truename()` und `falsename()` - Übersetzung von “true” und “false”
- `grouping()` - Groupierung der Tausenderstellen

Standardfacetten - *numpunct*

- Darauf aufbauend
  - `num_put` Facette, konvertiert numerischen oder booleanschen Wert in lokalisierten String
  - Definiert als:
    - `template<class charT, class InputIterator = istreambuf_iterator<charT> > class num_get`
  - Member
    - `OutputIterator put(OutputIterator s, ios_base& fg, char_type f1, long v) const`
      - `OutputIterator s` – Ort des Outputs
      - `ios_base& fg` – Format Flags (z.b. hex, dec, oct)
      - `char_type f1` – Füllzeichen für Whitespaces
      - `long v` – Wert zum Umwandeln, Überladungen für andere Basistypen existieren

Standardfacetten - *numpunct*

- `num_get` Facette, liest einen String ein und wandelt ihn in wert um
- Definiert als:
  - `template<class charT, class InputIterator = istreambuf_iterator<charT> > class num_get`
- Member:
  - `InputIterator get(InputIterator in, InputIterator end, ios_base& iob, ios_base::iostate& err, long& v) const;`
    - `InputIterator in, end` – Buchstabensequenz zum Umwandeln
    - `ios_base& iob` – Format Flags (z.b. hex, dec, oct)
    - `Iostate& err` – Fehleranzeige
    - `Long& v` – Referenz auf Speicher für Ergebnis, Überladungen existieren

numpunct, *num_get* Beispiel

```
const numpunct<char>& n_ger = use_facet<numpunct<char>>
(ger); // hole use_facet aus locale ger
const numpunct<char>& n_zh = use_facet<numpunct<char>> (zh);

cout.imbue(n_ger); // nötig, da sonst cout selber umwandlung
durchführt
cout << "true: " << n_ger.truename();
cout.imbue(n_zh);
cout << ", " << n_zh.truename() << endl;

// num_put facet
const num_put<char>& p_ger = use_facet<num_put<char>>
(ger);   // hole num_put facet
cout.imbue(ger);
p_ger.put (cout, cout, '0', 3.14159265); // benutze num_put
facet
cout << endl;

cout.imbue(zh);
use_facet<num_put<char>>(zh).put (cout, cout, '0', 3.14159265);
cout << endl;

cout.imbue(sys); // reset cout
```

Standardfacetten - *moneypunct*

- Wie numpunct:
  - `decimal_point(), thousands_sep(), do_grouping()`
- Außerdem:
  - `curr_symbol()` - Währungssymbol
  - `positiv_sign()` und `negative_sign()`
  - `fac_digits()` - Anzahl der Nachkommastellen
  - `pos_format()` und `neg_format()` - Formatierungsinfo:
    - 4er Array vom Typ `money_base::part: none, space, symbol, sign,value`
      - Deutsch: sign, value, space, symbol
      - Hong Kon: sign, symbol, value, non

Standardfacetten - *moneypunct*

- `money_put` Facette:
  - Konvertiert mit `put(...)` einen long double aus der kleinsten Einheit (z.b. 500 cent) in korrekt lokalisierten Wert
  - Nachkommastellen werden ignoriert
  - Ausgabestream braucht `ios_base::showbase` Flag, damit Währungssymbol und Punkte angezeigt werden
- `money_get` Facette
  - Konvertiert mit `get(...)` eine formatierte Zeichenkette zurück in einen long double

Standardfacetten - *time_put*

- `put(...)` gibt eine Zeitangabe mit dem Typ struct tm aus
  - Ausgabe mit einem printf-ähnlichen format-String, mit Syntax aus `strftime()`

specifier	Replaced by	Example
%a	Abbreviated weekday name *	Thu
%A	Full weekday name *	Thursday
%b	Abbreviated month name *	Aug
%B	Full month name *	August
%c	Date and time representation *	Thu Aug 23 14:55:02 2001
%d	Day of the month (01-31)	23
%H	Hour in 24h format (00-23)	14
%I	Hour in 12h format (01-12)	02
%j	Day of the year (001-366)	235
%m	Month as a decimal number (01-12)	08
%M	Minute (00-59)	55
%p	AM or PM designation	PM
%S	Second (00-61)	02
%U	Week number with the first Sunday as the first day of week one (00-53)	33
%w	Weekday as a decimal number with Sunday as 0 (0-6)	4
%W	Week number with the first Monday as the first day of week one (00-53)	34
%x	Date representation *	08/23/01
%X	Time representation *	14:55:02
%y	Year, last two digits (00-99)	01
%Y	Year	2001
%Z	Timezone name or abbreviation	CDT
%%	A % sign	%

* The specifiers whose description is marked with an asterisk (*) are locale-dependent.

Standardfacetten - *time_get*

- `InputIterator get_time(InputIterator begin, InputIterator end, ios_base& fg, ios_base::iostate& err, tm *t) const;`
- `InputIterator get_date(...)`
- `InputIterator get_weekday(...)`
- `InputIterator get_monthname(...)`
- `InputIterator get_year(...)`

- `dateorder date_order() const;`
  - Enumeration dateorder: no_order, dmy, mdy, ymd, ydm

time_pit - Beispiel

```
// time_put
basic_stringstream<char> s1, s2; // Nutzung statt cout
struct tm xmas = {0, 0, 12, 25, 11, 93};
const char* fmt = "Date: %c";

// put mit format-string
s1.imbue(ger);
use_facet<time_put<char>>(ger).put(s1.rdbuf(), s1, ' ', &xmas,
fmt, fmt+sizeof(fmt));

// put mit einem format-char
s2.imbue(zh);
use_facet<time_put<char>>(zh).put(s2.rdbuf(), s2, ' ', &xmas,
'c', 0);

cout << "German time: " << s1.rdbuf()->str() << endl;
cout << "Chinese time: " << s2.rdbuf()->str() << endl;
```