Standard C++ Locales

Angelika Langer

Trainer/Consultant

http://www.langer.camelot.de/

Agenda

- Introduction to I18N
- I18N Support in the C++ Standard Libary
- Creating and Accessing Locales
- Using Facets
- Adding User-Defined Facets

Cultural Differences

Alphabet

US: a-z A-Z & punctuation

German: as above & äöü ÄÖÜ ß

Greek: α–ω Α–Ω

Language

English

Deutsch

Français

Cultural Differences

Numbers

1,000,000.55

1.000.000,55

Currency

USD 10.00

$ 24.99

¥ 155

13,50 DM

Date

Sunday, March 3, 1996

Sonntag, 3. März 1996

Time

4:55 pm

16:55 Uhr

03:45:15

Sorting Strings

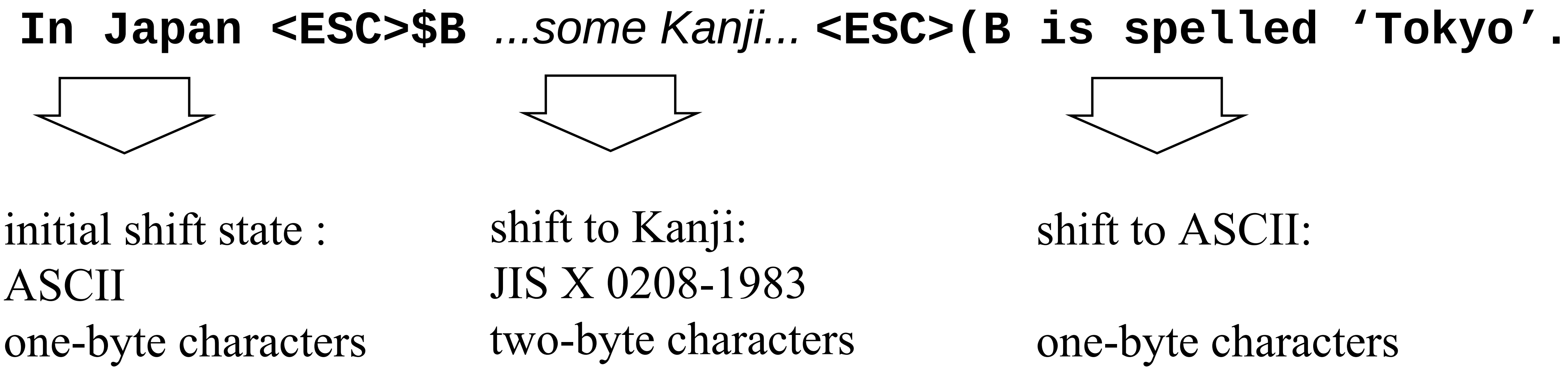
Sorted by ASCII rules	Sorted by German rules
Airplane	Airplane
Zebra	ähnlich
bird	bird
car	car
ähnlich	Zebra

Character Sets

- single-byte (7- or 8-bit)
  - 7-bit ASCII
  - 8-bit extensions of ASCII
    - additional characters, accented vowels, special symbols
    - Western European, Arabic, Greek, ...
- multi-byte codes
  - mixture of one and two-byte characters
    - Traditional Chinese, Kanji, ...

JIS Encoding

- requires *escape sequences* to shift between one- and two-byte modes.



`In Japan <ESC>$B` *...some Kanji...* **`<ESC>(B is spelled 'Tokyo'.`**

initial shift state :
ASCII
one-byte characters

shift to Kanji:
JIS X 0208-1983
two-byte characters

shift to ASCII:

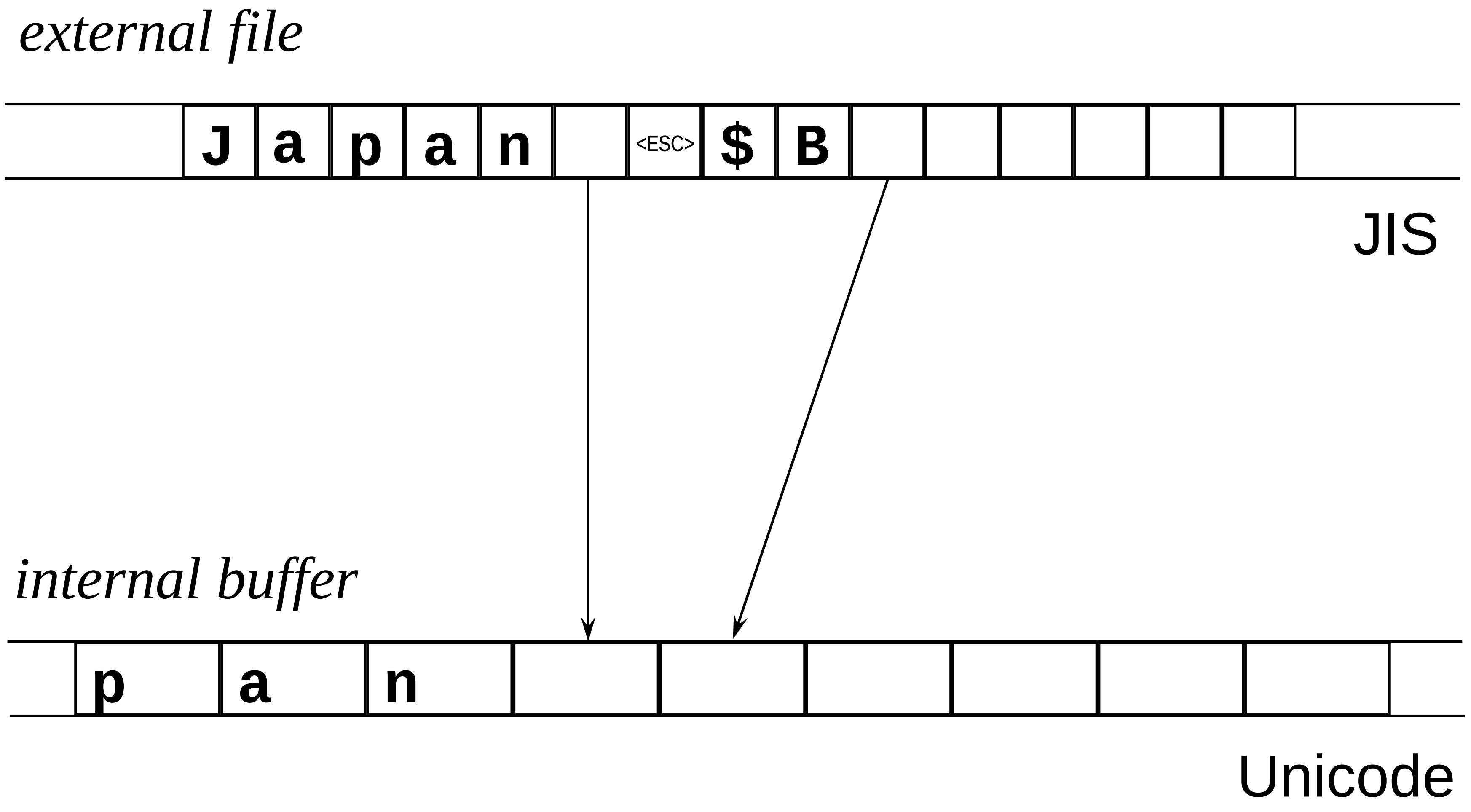
one-byte characters

Multi-Bytes vs. Wide Characters

- Multi-byte encodings
  - contain characters of different width,
  - are used on external media.

- Wide character sets
  - All characters have same size.
  - are used for in-memory representation.

Multi-Byte ↔ Wide Character Conversion



external file

JIS

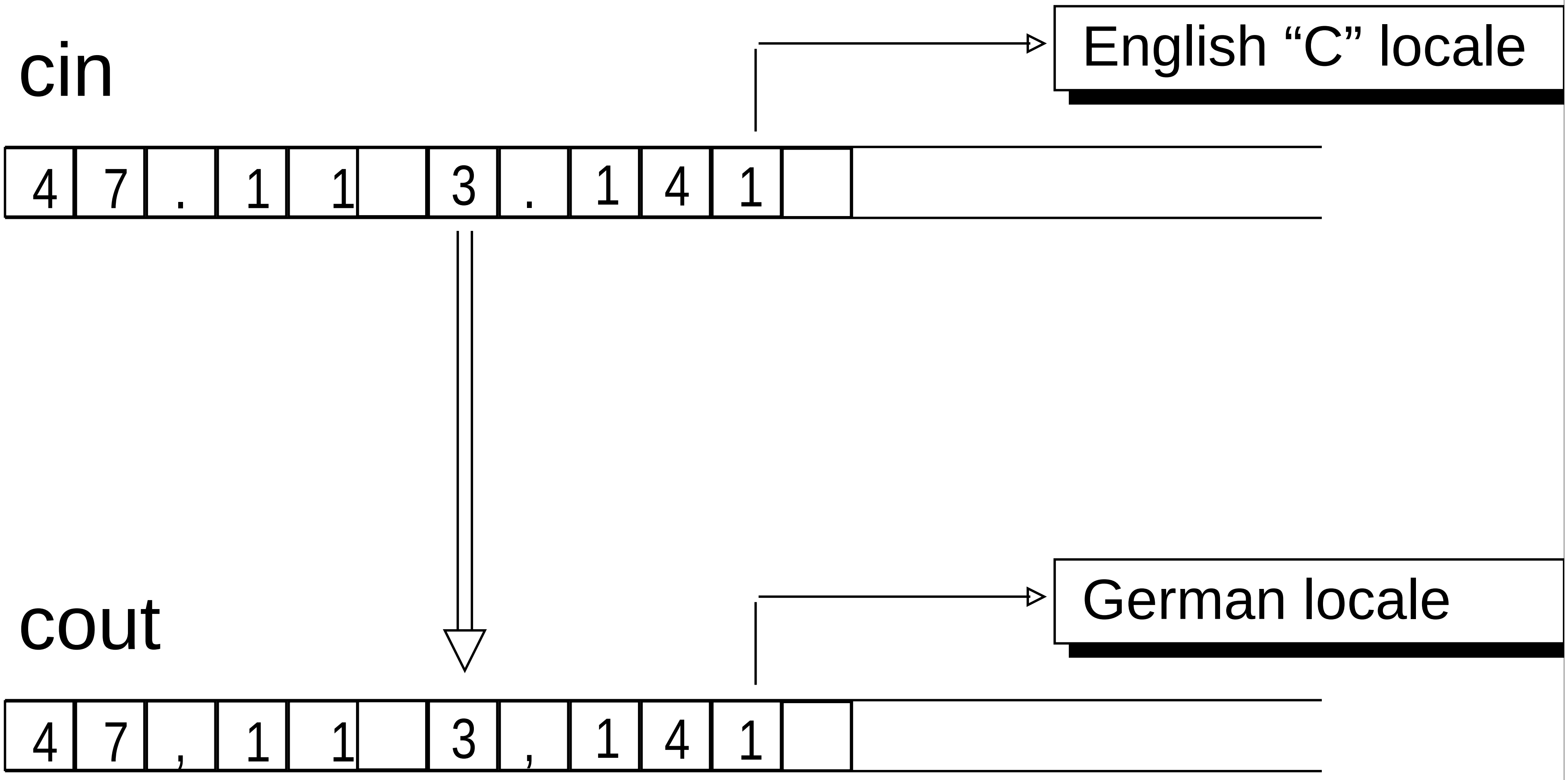
internal buffer

Unicode

Agenda

- Introduction to I18N
- I18N Support in the C++ Standard Library
- Creating and Accessing Locales
- Using Facets
- Adding User-Defined Facets

Using C++ Locales



cin

47.11 3.141

English “C” locale

cout

47,11 3,141

German locale

Using C++ Locales

```
cin.imbue(locale::classic());
cout.imbue(locale(“German”));
double f;
while (cin >> f)
   cout << f << endl;
```

```
Input:      47.11 3.141
Output:     47,11 3,141
```

Culture-Sensitive String Comparison

- `operator<()` for `basic_string<charT>` is not internationalized (performs lexicographical comparison of the character codes).
- For 'culture sensitive' string comparison the locale provides an overloaded function call operator `operator()()` :

```
template <class charT,class Traits,class Alloc>
bool operator()
(const basic_string<charT,Traits,Alloc>& s1,
 const basic_string<charT,Traits,Alloc>& s2)
```

Locales as Comparators

- Locale objects can be use as a comparator with standard containers and algorithms.

```
locale German(“German”);
map<string,long,locale> phoneDir(German);
```

```
locale German(“German”);
vector<string> names;
sort(names.begin(),names.end(),German);
```

Facets and Locales

- Internationalization services bundled into so-called *facets.*
- A facet
  - encapsulates data that represents a set of culture and language dependencies and/or
  - offers a set of related internationalization services.
- A *locale* is a container of facets.
  - Locales are objects of class type called `locale` and facets are objects of a facet type derived from `locale::facet`.

Facet Types

Facet types are either

- predefined in the standard library (standard facets) or
- user-defined.

S*tandard facets*

- cover the basic set of cultural differences
- are automatically contained in every locale

User-defined facets

- cover further areas of cultural differences
- only present in a locale, if they were explicitly added

The Standard Facets

numeric	num_get<charT,InputIterator> num_put<charT, OutputIterator> numpunct<charT>	1.000,00 1,000.00
monetary	money_get<charT,InputIterator> money_put<charT,InputIterator> moneypunct<charT,bool International>	$ 100.00 100,00 DM
time	time_get<charT,InputIterator> time_put<charT,OutputIterator>	5:00 pm 17:00 h 31.01.95 01/31/95

The Standard Facets

ctype	ctype<charT>	isspace()
		tolower()

collate	collate<charT>	a,u,o,n,c
code conversion	codecvt<fromT,toT,stateT>	wide char multibyte

messages	messages<charT>	open(cat)
		get(msgid)

Facets

- ◆ Each facet offers a set of internationalization services.

```
template <class charT, class InputIterator>
class time_get : public locale::facet  {
public:
  iter_type get_time(iter_type s, iter_type end,
       ios_base&, ios_base::iostate& err, tm*)  const;
  iter_type get_date(...)  const;
  iter_type get_weekday(...) const;
  iter_type get_monthname(...) const;
  iter_type get_year(...) const;
  };
```

Agenda

- Introduction to I18N
- I18N Support in the C++ Standard Library
- Creating and Accessing Locales
- Using Facets
- Adding User-Defined Facets

Creating Locale Objects

A locale object is created either by:

- – providing a locale name,
- – combining two existing locales, or
- – combing an existing locale with an existing facet.

- The default constructor creates a snapshot of the current global locale.

Named Locales

Locale names

- same names as in the standard C library

`"C"`: classic US English ASCII locale

- default; implicitly used if programs is not internationalized
- created saying `locale("C")` or calling static function `locale::classic()`
-

`""`: native locale configured for a system

C locale names: syntax and semantics implementation-specific

- `"De_DE"` on X/Open same as `"German_Germany.1252"` on Microsoft

Combined Locales

- cannot add or replace facets in an existing locale object

- locale objects are immutable
  - their content does not change during their lifetime
  - None of the contained facets can be modified or replaced, nor can facets be added or removed from a locale.

- non-standard locales can only be created as a copy of an existing locale
  - with one or several facets replaced or added

Creating Combined Locales

```
template <class Facet>
locale combine(const locale& other);
```

- creates a copy of the locale object it is invoked on, and the copy has the facet of type `Facet` replaced or added by the corresponding facet from the existing locale `other`

```
locale holland("Dutch");
dutch_german
= locale("German").combine< moneypunct<char> >(holland);
```

Retrieving Facets

- `template <class Facet>`
  `bool has_facet(const locale&) throw()`
  - allows to check whether a facet of the specified facet type is contained in the specified locale

- `template <class Facet>`
  `const Facet& use_facet(const locale&)`
  - returns a reference to the contained facet, if present, and throws a `bad_cast` exception otherwise

Retrieving Facets

- When these functions are invoked, the template argument (i.e. facet type) must be explicitly specified.

```
locale loc;    // snapshot of the current global locale

if (has_facet< money_put<char> >(loc)
   const money_put<char>& fac1
   = use_facet< money_put<char> >(loc);

if (has_facet< money_put<char,string_inserter<char> > >(loc))
   const money_put<char,string_inserter<char> >& fac2
   = use_facet< money_put<char,string_inserter<char> > >(loc);
```

use_facet<Facet>()

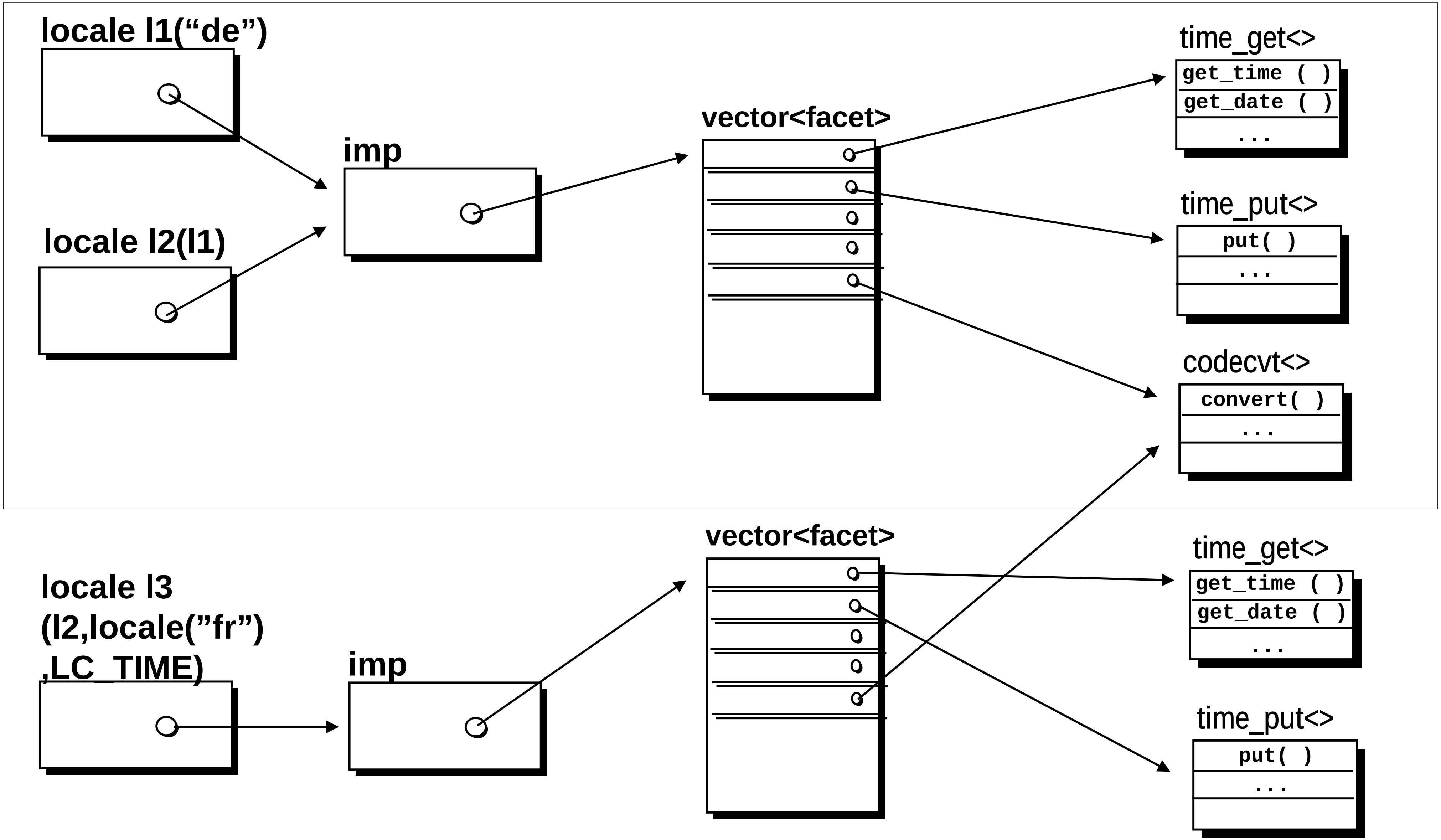
`use_facet<Facet>(loc)`

- returns a reference to the requested facet, if found
- throws a `bad_cast` exception otherwise

How long does the reference stay valid?

- at least as long as any copy of the containing locale exists

Architecture of C++ Locales



Temporary Locale Objects

Do NOT create any temporary locale objects.

- The validity of the facet reference is tied to the lifetime of its containing locale and any copies of that locale, and
- might become invalid before its use, because the containing locale has already been destroyed.

```
const numpunct<char>& fac
= use_facet<numpunct<char> >(locale("German"));

// program crash:
cout << "true in German: "
     << fac.truename() << endl;
```

Agenda

- Introduction to I18N
- I18N Support in the C++ Standard Library
- Creating and Accessing Locales
- Using Facets
- Adding User-Defined Facets

Facet Families

A ***facet family*** is a hierarchy of facet types that are derived from each other.

- base class defines the family’s facet interface

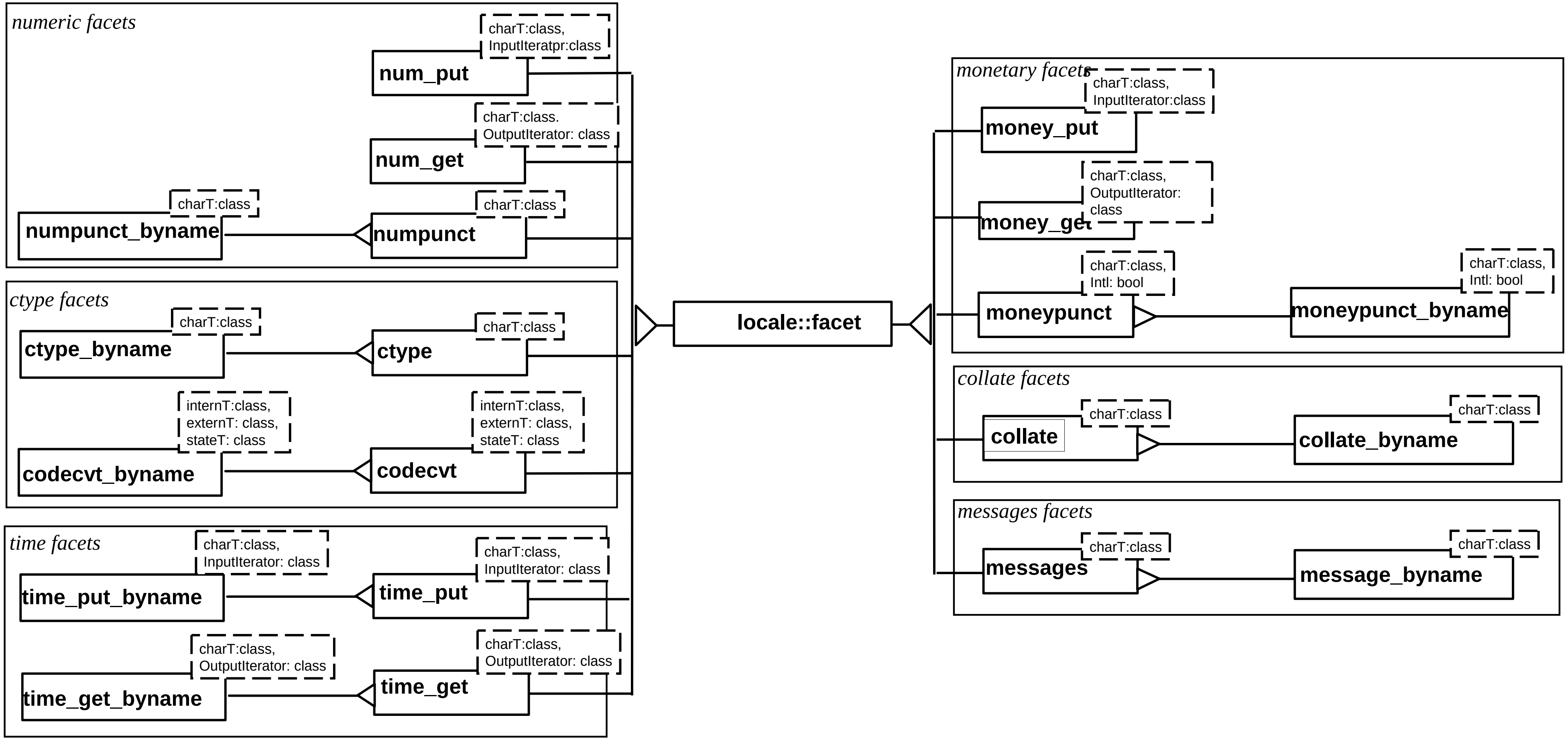
Some facet families are closely related:

- base classes created from a facet base class template

Example:

- base class template of the ctype facet families
  `template <class charT> class ctype`
- facet base classes (instantiations or specializations)
  `ctype<char>` and `ctype<wchar_t>`
- family members (derived classes)
  `ctype_byname<char>` and `ctype_byname<wchar_t>`

Facet Families



numeric facets

num_put
charT:class,
InputIteratpr:class

num_get
charT:class.
OutputIterator: class

numpunct_byname
charT:class

numpunct
charT:class

ctype facets

ctype_byname
charT:class

ctype
charT:class

codecvt_byname
internT:class,
externT: class,
stateT: class

codecvt
internT:class,
externT: class,
stateT: class

time facets

time_put_byname
charT:class,
InputIterator: class

time_put
charT:class,
InputIterator: class

time_get_byname
charT:class,
OutputIterator: class

time_get
charT:class,
OutputIterator: class

locale::facet

monetary facets

money_put
charT:class,
InputIterator:class

money_get
charT:class,
OutputIterator:
class

moneypunct
charT:class,
Intl: bool

moneypunct_byname
charT:class,
Intl: bool

collate facets

collate
charT:class

collate_byname
charT:class

messages facets

messages
charT:class

message_byname
charT:class

Locales and Facets

- Facets are rarely used stand-alone (i.e. independently of a locale).
  - Usually, all facets relevant for a certain cultural area are bundled into a locale object.

➢ Each locale object contains at most one facet from a given facet family.

➢ Facets in a locale can be identified by means of their family name (base class type).

Advanced Usage of Standard Facets

There are several ways of using facets, depending on how they are maintained:

- Indirect Use of a Facet Through a Stream
- Use of a Facet Through a Locale
- Direct Use of the Facet Independently of a Locale

Use of Facets Through Streams

- Each stream has a locale attached.
- Various stream operations use standard facets contained in the stream's locale for performing their tasks.
  - code conversion facets for converting between internal and external character encodings
  - ctype facets recognition of whitespace character, digits, etc. during parsing
  - numeric facets used by the inserters and extractors for numeric values
- Inserters and extractors offer a convenient way of using the facets' capabilities.

Internationalized Number Formatting

- Attach the desired locale to a string stream, write the numeric value to the string stream, and afterwards extract the resulting string from the string stream.

```
ostringstream ost;
ost.imbue(locale("German"));
ost << setprecision(2) << uppercase << scientific;
ost << 831.0 << ‘ ‘ << 8e2;
string s = ost.str();
```

- Afterwards the string `s` contains:

```
"8,31E+02 8,00E+02"
```

Use of Facets Through Streams

Use of formatting and parsing facets through a stream is the most convenient way of using these facets.

Internationalized parsing and formatting of

- numeric values is available through the stream classes via the predefined inserters and extractors.
- date and time values is not available through the stream classes.
  - There are no standard types for representing date and time values.
  - Such inserters and extractors can be added.
- other values can be handled in the exact same way.
  - Define a facet type for address formatting rules, install such facets in a locale, attach that locale to a stream, define an inserter for address values uses the address formatting facet.

Use of Facets Through Locales

Write the result of formatting of a numeric value to a string object of type `string`.

- use the num_put facet's `put()` function, which writes to an character container via an output iterator

```
template<class charT,
         class OutputIterator
         = ostreambuf_iterator<charT> >
class num_put
```

  - generates a formatted character sequence from a numeric or Boolean value

Use of Facets Through Locales

- must provide an iterator that allows output to the string
  - prefer an insert iterator of type `back_insert_iterator <string>` over a plain string iterator of type `string::iterator`, in order to make sure that the string grows as needed

- need a num_put facet of type `num_put<char, back_insert_iterator<string> >`
  - no locale contains such a facet
  - we must explicitly install it in the locale object that we want to use

The `num_put` Facet

```
OutputIterator
put(OutputIterator s, ios_base& fg,
    char_type fl, double v)
```

parameters:

- an output iterator
  - location to which the formatted string should be written
- a reference to an `ios_base` object
  - to retrieve information contained in numpunct facet in the locale attached to the `ios_base` object
  - to retrieve format flags contained in the `ios_base` object
- a fill character
  - used for padding
- the value to be formatted

Internationalized Number Formatting

```
typedef num_put<char,back_insert_iterator<string> >
        string_num_put;

locale loc(locale("German"), new string_num_put);

basic_ios<char> str(0);
str.imbue(loc);
str.precision(2);
str.setf(ios_base::uppercase|ios_base::scientific);

string s;
back_insert_iterator<string> iter(s);

const string_num_put& fac = use_facet<string_num_put>(loc);

iter = fac.put(iter,str,' ',831.0 );
*iter++ = ‘ ‘;
iter = fac.put(iter,str,' ',8e2);
```

Use of Facets Through Locales

- significantly less convenient than use through streams

- worst-case example
  - other facets are easier to use independently of streams
  - examples:
    - collation through locale’s function call operator
    - character classification through global functions like `isspace(char,locale)`, etc.

- facets tightly coupled to streams:
  - parsing and formatting facets for numeric, monetary, and time/date values
  - code conversion facets

Use of Facets Without Locales

Facets are designed to be contained in locales.

- All facet types have a protected destructor.
- Objects of a type with an inaccessible destructor can only be created on the heap, hoping that someone who has access to the destructor will eventually delete the heap object.
- That is exactly, what facets are designed for:
  - we create them on the heap and
  - hand them over to a locale, which is a friend of all facet types and has access to the protected destructor, and
  - the locale deletes the facets, once it will not be used any longer.

Do we have to stuff facets into locales?

It looks kind of stupid to stuff the facet into a locale first, and then retrieve it again so that it can be used. Why did we do it?

- The num_put facet needs other facets.
- Stuffing all of the facets into one locale object makes it easy to pass around all the necessary information in form of the locale object.
- Still, we can do it differently. A facet need not necessarily be contained in a locale.

Stand-Alone Facets

If we want to use a facet independently of a locale, then we need an additional abstraction that allows to create and destroy facet objects.

We wrap the original facet in a derived class that has an accessible destructor:

```
template <class Facet>
class StandAloneFacet
 : public Facet
{
 public:
    StandAloneFacet() : Facet(1) {}

    ~StandAloneFacet() {}
};
```

The `StandAloneFacet` Wrapper

- simple wrapper around the actual facet
- derived from the facet type that it encapsulates
- provides the missing public destructor
- base class constructor called with the value `1` as an argument
  - indicates that the facet is used stand alone, i.e. the memory is correctly managed by the base class

Internationalized Number Formatting

- create a wrapper, provide an `ios_base` object with format flags and attached locale, and call the facet’s `put()` function

```
typedef num_put<char,back_insert_iterator<string> >
        string_num_put;
StandAloneFacet<string_num_put> fac;

basic_ios<char> str(0);
str.imbue(locale("German"));
str.precision(2);
str.setf(ios_base::uppercase|ios_base::scientific);

string s;
back_insert_iterator<string> iter(s);

iter = fac.put(iter,str,' ',831.0 );
*iter++ = ‘ ‘;
iter = fac.put(iter,str,' ',8e2);
```

Agenda

- Introduction to I18N
- I18N Support in the C++ Standard Library
- Creating and Accessing Locales
- Using Facets
- Adding User-Defined Facets

User-Define Facet Types

[1] Facet types must be subclasses of class `locale::facet`.

[2] They must contain a *facet identification* in form of a static data member that is declared as

```
static locale::id id;
```

- The identification is used for maintenance and retrieval of facets from a locale and
- identifies an entire family of facets:
    - All facets with same identification belong to same facet family.
    - A locale cannot contain two facets with identical identification.
    - Facets from the same family replace each other.

User-Defined Facet Types

New types of facets can be added

- by deriving from existing facet types, in which case the facet identification is inherited and the new facet belongs to an already existing facet family, or
- by defining a new facet class that has a facet identification of its own, in which case a new facet family is introduced.

Adding to an Existing Facet Family

Character Classification for Umlaut

- The German alphabet includes so-called umlaut characters; these are `'ä'`, `'ö'`, `'ü'`, `'Ä'`, `'Ö'`, and `'Ü'`.
- We want to provide an extended ctype (character classification) facet that can identify umlaut characters.
- The new facet type shall belong to the ctype facet family and must be derived from one of the ctype facet types.

```
template <class CharT>
class umlaut : public ctype_byname<CharT> {
public:
   explicit umlaut(size_t refs);
   bool is_umlaut(CharT c) const;
};
```

Implementing the Umlaut Facet

```
template <class CharT>
class umlaut : public ctype_byname<CharT> {
public:
   explicit umlaut(size_t refs = 0)
   : ctype_byname<CharT>("German",refs) {}

   bool is_umlaut(CharT c) const { return do_is_umlaut(c); }
protected:
   virtual bool do_is_umlaut(CharT c) const
   { switch(narrow(c))
     { case 'ä': case 'ö': case 'ü':
       case 'Ä': case 'Ö': case 'Ü': return true;
       default:                      return false;
     }
   }
};
```

Using the Umlaut Facet

```
locale loc(locale("German"), new umlaut<char>);

if (has_facet<umlaut<char> >(loc))
{ const umlaut<char>& ufac = use_facet<umlaut<char> >(loc);
  cout << ufac.is(ctype_base::alpha,'Ä') << endl;
  cout << ufac.is_umlaut('Ä') << endl;
}
const ctype<char>& cfac = use_facet<ctype<char> >(loc);
cout << cfac.is(ctype_base::alpha,'Ä') << endl;
cout << cfac.is_umlaut('Ä') << endl;  // error
```

- When the `umlaut` facet is retrieved via its actual derived class type, then the `is_umlaut()` function is accessible.
- If we use the `umlaut` facet as an ordinary ctype facet and retrieve it by its base class type, then only the ctype facet interface is accessible and `is_umlaut()` cannot be invoked.

Defining a New Facet Family

How can internationalization services that have no relationship to any of the existing facets be bundled to a new facet interface and implemented as a new facet family?

Facet Base Classes (recap):

- Each facet base class has a facet identification of its own.
- Typically there is an entire hierarchy of facet classes,
  - that inherit and optionally override the facet base class’s interface.
- All facet types in such a hierarchy form a facet family.
  - all family members have the same facet identification
- A locale object contains exactly one representative from that facet family.

Address Formatting Facet Family

Concrete example: a facet interface for formatting of international addresses

- define a facet base class that has a new facet interface for address formatting and a new facet identification
- build two derived address formatting facets
- demonstrate how they can be used in conjunction with IOStreams for implementation of an address inserter
- explore how the installation of an address formatting facet in a locale object could be automated and
- suggest a locale factory for that purpose

International Address Formats

German address pattern

<FirstName> <LastName>
<Address1>
[<Address2>]
<blank line>
[<CountryCode>-]<PostalCode> <City>

example

Dorothea Meier
Krickelberg 5

D-41836 Ratheim

International Address Formats

US address pattern

<FirstName> <MiddleInitial> <LastName>
<Address1>
[<Address2>]
<City>, <State> <PostalCode>
[<Country>]

example

Dorothea S. Meier
1 W Superior Place
Chicago, IL 60610
U.S.A.

The Address Class

```
template<class charT> class address {
public:
 typedef basic_string<charT> String;

 address(const String& firstname, const String& secname,
         const String& lastname,
         const String& address1,  const String& address2,
         const String& town,      const String& zipcode,
         const String& state,     const String& country,
         const String& cntrycode);

 string firstName();
 ...
private:
 ...
};

basic_ostream<charT>&
operator<<(basic_ostream<charT>& os,const address<charT>& ad);
```

The Address Formatting Facet

- define a new facet family for address formatting
  - by building a new facet type with an identification of its own
- following the naming conventions of the standard:
  - name the address formatting facet `address_put`
  - the formatting operation is a member function called `put()`
- use output iterators
  - to designate the target location of the formatted address string
  - make the address facet a class template taking the output iterator type as a template argument
- use delegation to virtual protected interface
  - the public interface consists of non-virtual member functions that delegate all tasks to protected virtual member functions

The Address Formatting Facet

```
template<class charT,
         class OutIter = ostreambuf_iterator<charT> >
class address_put : public locale::facet {
  typedef basic_string<charT> String;
public:
  typedef OutIter iter_type;
  static locale::id id;

  address_put(size_t refs = 0) : locale::facet(refs) {}

  void put(OutIter oi, const address& addr) const;

protected:
  virtual void do_put (OutIter oi,
                       const address& addr) const;
};
```

Facets for Concrete Cultural Areas

What turns our address facet into a German or a US address facet?

- For many of the standard facets, there are byname versions that accept the name of a localization environment as a constructor argument.
- To keep our example focused, we derive an address facet for each specific cultural area from the base class template `address_put`.

A US Address Facet

```
template<class charT,
         class OutIter = ostreambuf_iterator<charT> >
class US_address_put : public address_put<charT, OutIter> {
public:
  US_address_put(size_t refs = 0)
  : address_put<charT,OutIter>(refs) {}
protected:
  virtual void do_put(OutIter oi,
                      const address& addr) const
  {String s(addr.firstName());
   s.append(" ").append(addr.middleInitial()).append(" ").
     append(addr.lastName()).append("\n");
   ...
   put_string(oi,s);  // helper function; see next slide
  }
};
```

Helper Function

The helper function `put_string()` writes the formatted string to the output iterator.

```
template<class charT,
         class OutIter = ostreambuf_iterator<charT> >
class address_put : public locale::facet {
  // ...
protected:
  void put_string(OutIter oi, String s) const
  {typename String::iterator si, end;
   for (si=s.begin(), end= s.end(); si!=end ; si++, oi++)
       *oi = *si;
  }
};
```

The Address Inserter

```
template <class charT>
basic_ostream<charT>&
operator<< (basic_ostream<charT>& os,
            const address<charT>& addr)
{
 locale loc = os.getloc();
 try {
   const address_put<charT>& apFacet
         = use_facet<address_put<charT> > (loc);
   apFacet.put(os, addr);
 } catch (bad_cast&)
 { /* locale does not contain a address_put facet */ }
 return (os);
}
```

Equipping Locales with Address Facets

- Equip a standard locale with an additional address formatting facet.

```
locale usLocaleWithAddressPut
       (locale("En_US"), new US_address_put<char,osIter>);
```

- Construction of a locale object with additional facets of user-defined types (a *non-standard* facet) involves:
  - retrieval or creation of a standard locale object for the cultural area,
  - retrieval or creation of the additional non-standard facet(s) for that area, and
  - combining both to a new, extended non-standard locale object.

A Locale Factory

Decouple the process of locale construction from locale use.

- build a factory that handles the construction of locale objects
- create locale objects "byname":
  - they shall have all standard facets for the cultural area specified by the name,
  - plus a number of desired, additional non-standard facets, like an address formatting facet for instance
- build a hierarchy of locale factories:
  - a base locale factory creating standard locale objects and
  - derived factories for non-standard locales

Base Locale Factory

```
class locale_factory {
public:
   virtual locale make_locale (const char* name) const
   { return locale(name); }
};
```

Remark:

- Usually a factory returns a pointer or reference to the created object.
  - derived factories must be allowed to create objects of derived classes, which can have additional members or vary in the behavior of existing member functions
- Our factory returns a locale *object* rather than a pointer or a reference.
  - locales are passed around as objects
  - internally only a handle to an arbitrary number of facets from arbitrary facet families

Concrete Locale Factory

- uses the `map` container from the standard library for mapping a locale name to the respective `address_put` facet, so that non-standard locale objects can be created

- returns a locale containing all standard facets and, if a US or a German locale is requested, additionally an `address_put` facet

Concrete Locale Factory

```
class address_locale_factory : public locale_factory {
 typedef ostreambuf_iterator<char> osIter;
public:
 address_locale_factory()
 { facets["En_US"] = new US_address_put<char,osIter>(1);
   facets["De_DE"] = new DE_address_put<char,osIter>(1);
   ...
 }
 ~address_locale_factory()
 { delete facets["En_US"];
   delete facets["De_DE"];
   ...
 }

 locale make_locale (const char* name) const;

private:
 map<string, address_put<char,osIter>* > facets;
};
```

Concrete Locale Factory

```
class address_locale_factory : public locale_factory {
public:
 address_locale_factory();
 ~address_locale_factory();

 locale make_locale (const char* name) const
 { if (facets.find(name) == facets.end())
      return // name unknown; make standard locale
            locale_factory::make_locale(name);
   else
      return // make extended locale
            locale(locale_factory::make_locale(name),
                   (*(facets.find(name))).second);
 }
private:
  map<string, address_put<char, osIter>* > facets;
};
```

Putting the pieces together

```
void printAddress(ostream& os,
                  const address<char>& address,
                  locale loc)
{
   locale original = os.imbue(loc);
   os << address << endl;
   os.imbue(original);
}
```

- A locale that has an address facet installed, must be provided on invocation:

```
printAddress
(cout,
 myAddress,
 address_locale_factory().make_locale("German")
);
```

User-Define Facets

Mandatory. A user-defined facet type must

- be derived from class `locale::facet` and
- have a facet identification in form of a static data member named `id` of type `locale::id`.

Recommended.

- A facet name should follow the naming conventions of the standard facets.
- Formatting and parsing operations should access source or destination via iterators.
  Formatting and parsing facets should be templatized on the iterator type and use stream buffer iterators as a default.
- Public member function should delegate to protected member functions.

Wrap-Up

- Locales are containers of facets.
  - responsible for memory management and retrieval of facets
- Facets are bundles of related internationalization services and information.
  - designed for use in conjunction with a locale
- Use of I18N services is usually through
  - streams (for parsing and formatting of text representations) or
  - convenience functions
- C++ standard locales
  - ready-to-use services in form of standard facets
  - framework to be extended by user-defined facets

Wrap-Up

- unusual design
  - access to facets through their base type
- advantage
  - extremely flexible
  - facet interfaces are not restricted in any way
  - still the locale can maintain them no matter what type the are of
  - still it’s type-safe; facets are retrieved via their actual type

Recommended Reading

Angelika Langer & Klaus Kreft
Standard C++ IOStreams and Locales
Addison Wesley, January 2000

David Schmitt
International Programming for Windows
Microsoft Press , April 2000

Bjarne Stroustrup
The C++ Programming Language, Special Edition
Addison Wesley, January 2000

Nicolai Josuttis
The C++ Standard Library
Addison-Wesley, July 1999

Contact Info

Angelika Langer

Training & Mentoring

Object Oriented Software Development with C++ and Java

email: langer@camelot.de

http://www.langer.camelot.de/