Doc: WG21/N0860=X3J16/96-0042
Project: C++ Standard Library
Date: Jan 1996
Reply to: Nathan Myers
<ncm@cantrip.org>

Locale Architecture

Most people (myself included!) would prefer to ignore locales. The C++ locale design supports this preference, but that support is in danger. To continue safely to ignore locales you will need, unfortunately, to pay attention to this issue.

In Tokyo, a proposal that came dangerously close to passing would have eliminated the encapsulation designed into the Clause 22 locale components, making them essentially unusable as objects. I expect that proposal may surface again in Santa Cruz. This paper explains the reasoning behind the design of the locale components. I hope it will then become obvious why the change would be so destructive. This document also proposes some non-destructive alternatives.

History

The story begins long ago, during C standardization. Plauger describes the origin of C’s <locale.h> header:

> This particular header popped up about five years after work began on the C Standard. ... About then, we learned that a number of Europeans were unhappy ... [They] took it for granted that an ISO standard for C must differ from the ANSI C Standard. ... So we asked the Europeans to show us their shopping list of changes. ... The machinery eventually adopted is remarkably close to the original proposal. ... Many of the objections to ANSI C ... were derailed. ... WG14 is still working on additions to the existing C Standard.

The standard C library’s locale facilities were proposed late and hastily adopted under threat from outside the committee, but were still unsatisfactory. How does this affect us today?

The fact is that the C locale is used very rarely, and where it is used, it is used very shallowly, even in Europe where it should have been most useful. It is used so little that many, even in Europe, have come to believe that standardized locale facilities are necessarily unusable.

Yet foreign markets are recognized as strategically important. Hence, each large company has a few internationalization experts who attend internationalization conferences, and evangelize locale use among others in their company. Those others recognize the subject as quicksand, and avoid it much as possible. (Don’t you?)

One belief common (but not universal) among these experts is that a fundamental lesson of software development -- "Encapsulation good, global data evil" -- doesn’t apply where internationalization is involved. The standard C library locale components are relentlessly global, so almost everyone who has coded internationalization in C has experience only with global locales. Many have come, through familiarity, to believe that it is a reasonable approach.

The C++ Locale

The ISO/ANSI C++ Standard effort offered an opportunity to start afresh, and create something usable. We had several advantages:

TABLE 1

- o We have the benefit of hindsight;
- o We have a core language of far greater expressive power than C;
- o Because we had a mandate from the beginning, we can afford to think architecturally and design carefully;
- o We are inherently suspicious of arguments against encapsulation and in favor of dependence on global data;
- o Because we are not touching the C locale, anything we do cannot fail to satisfy those who (for whatever reason) actually prefer the C locale.

To begin a fresh design, we need a "shopping list" not of changes from C, but of basic requirements. Here are some of the goals the C++ locale is designed to meet, in order of importance:

TABLE 2

Encapsulated:
The C++ locale must be ignorable by anyone who doesn’t want to use it. Use of it in one part of a program must not require knowledge of that use in other parts of the program.

Extensible:
Any standard list of locale facilities (number formats, date formats, time zone rules) is necessarily far from complete, so locales must be extensible in several dimensions by end users without compromising type safety or system integrity.

Safe:
Locales objects must manage their own memory. The interface to locale components must not offer easy opportunities for undetectable user errors, or require casts.

Convenient:
Use of one feature of a locale must not require learning about and operating other parts. Use of locale components must be easily hidden behind higher-level interfaces.

Object-based:
Locales will be used in distributed programs, where the "local preferences" vary from one part of the program to another; also, even single users often have reason to use more than one format or encoding. Hence, it must be practical and easy to use two, or many, locales simultaneously.

Re-entrant:
Locales will be used in multi-threaded programs, so the design must contain no interfaces that preclude efficient re-entrant implementation.

Isocapable:
The C++ locale must not lack features found in the C locale.

(However, bugs in the C locale design must be left behind.)
It must not depend on more environmental resources than the C
library.

Non-multibyte:
Multibyte character encodings are sometimes prohibitively hard to
use, so the C++ locale must make it easy to convert to fixed-size
characters at the boundaries of a system.

The Design

The most demanding requirement, by far, is encapsulation. It affects
everything about the design, often subtly. Extensibility has a more
visible effect. The remaining requirements are not difficult once the
first two are satisfied.

We begin, of course, with an object. For safety and convenience,
locale objects are values -- we can copy them cheaply, compare them
for equality, store them compactly -- the usual nice properties.

A locale object resembles a container, a map, but indexed by type, at
compile time. The indexing operator, therefore, is not operator[],
but rather the template operator <>. The objects contained, Facets,
are all derived from class locale::facet, which both manages memory
and helps ensure type safety. (Standard facets, too, are extensible,
by derivation, but this paper focuses on the core semantics of locale
objects.) The result is that access to a facet of type Timezone in
locale loc is accessed using the syntax:

```
  use_facet<Timezone>(loc)
```

This resembles a (new-style) cast both in syntax and semantics.
The result is a reference to a Timezone object, so Timezone member
functions may be called:

```
  use_facet<Timezone>(loc).isdaylight(now);
```

Returning a reference to internal data seems dangerous, and without
carefully designed guarantees it would be intolerable. An interface
that generates dangling references would be everybody’s nightmare.

Before exploring these guarantees, let us consider alternatives.
The template use_facet<>() need not have returned a reference; it
could have returned something like a smart pointer. This approach
ignores, however, that facets in a locale are interdependent --
formats depend on the choice of character encoding, for instance.
Facets cannot be used, practically, in isolation from their locale.

Another alternative was to throw caution to the wind; provide no
guarantees to the user of the locale facet, and place all the
responsibility on the locale provider not to change anything
that might be depended on. This, however, creates an impossible
restriction on the provider of the locale, because it requires
knowledge of how all other parts of the program are using the
locale. It breaks encapsulation, just like global data.

What guarantees are necessary to make internal references safe?
The most important requirement on the guarantees is that they be
easy for the user of a locale to understand. The simplest approach
is to tie the lifetime of the reference to the lifetime of (copies
of) the locale value it was extracted from. This requires, then,
that it be easy to track the lifetime of the locale value. The

simplest way to ensure this is make locales immutable. Thus, you can create a locale object, and assign it, but all other operations are const; then the only operation to watch out for is assignment.

To make all this more concrete, imagine a user function that uses a locale obtained from an unknown source. It extracts references to various facets, and calls member functions of those facets to construct local context. (For example, it might convert the source character set digits 0..9 to corresponding characters in the preferred encoding, and construct a state machine to use in parsing.)

If these references were to become invalid because the locale changed without notice, then even if a reference were not stored the saved results would become obsolete, and probably inconsistent with the rest of the locale. This kind of inconsistency leads to program crashes. Thus, lifetime of references returned is a shorthand for lifetime of valid results from members called on those references.

The Locale Invariant

The above reasoning leads us to the Locale Invariant: use_facet<F>(loc) for some *F* and any copy of *loc* always returns the same reference. This protects against dangling references, and against inconsistency between references obtained for two facets F1 and F2 called at different times.

The invariant can be satisfied in many ways, and some are outlined in the open issue 22-037 of the Clause 22 Issues List. The simplest is to say that any facet requested that is not found in the locale is not provided, and an exception is thrown. (I now believe this would be the best approach.)

A less strict approach would be to say that any facet found in the global locale at the time a locale is constructed, and not represented in that locale, is "adopted", and becomes part of the locale. This is an easy way to propagate user-defined facets safely around a system.

A third approach, which is the statusquo, is to say that if the facet is not present when requested, it is at that point "adopted" from the global locale. This is what has been called "caching", and which all parties have agreed is undesirable.

An Unwise Attempt at Compromise

This third approach was proposed as part of an attempt at a compromise with users of who prefer locales based on global data. Another part of this compromise was the provision for a special "transparent" locale for which the invariant is utterly violated. For this compromise to be tolerable to users of locale objects, it was necessary to restrict these broken "transparent" locales from anywhere they can do harm, such as in iostream.

The attempt at compromise failed, in that those it was intended to satisfy have wholly rejected it, and attacked the Locale Invariant itself. It would substantially simplify locale to eliminate the compromise, and (if necessary) pursue some other approach. That is: strike the transparent locale and strike the "adoption" semantics. As we have National Body comments describing the status quo as unacceptable, we have cause to proceed with this simplification. If a different compromise can be proposed by those who prefer C semantics but are dissatisfied with the C locale, it can be proposed and considered on its own merits.

What, then, shall we say to those who want to imbue "transparent" or "partially transparent" locales into iostreams, regardless of the consequences? They argue that compatibility with "past practice" requires some concession. Past practice, however, is the Standard C Library. It remains untouched, and it remains exactly as usable as ever.

Conclusion

The reader should notice that none of the arguments above depend on references to multithreading, or to performance of or convenience in implementing the Standard Library. While such arguments present useful examples, the issue is more fundamental, and concerns all potential users of locales, even those who want to know nothing about the subject.

If the Locale Invariant were to be broken for the sake of "simplicity", we could gain far greater simplicity at idential cost in expressive power usability simply by eliminating locale objects entirely, and returning to a single global locale.

Proposed Resolution:

One of:

1. Eliminate the Failed Compromise

Simplify the semantics of locale members, and of locale use by striking out the static member locale::transparent(), and all mention of it. Change the description of use_facet<> as follows:

```
  template <class Facet>
    const Facet& use_facet(const locale& loc);

  Get a reference to a facet of a locale.
  Returns: a reference to the corresponding facet of loc, if present.
  Throws: bad_cast if the facet is not present.
  Notes: the reference returned remains valid at least as long as any
    copy of loc exists.
```

This results in the cleanest, simplest possible semantics, and eliminates "caching".

2. Patch the Failed Compromise

If (1) is considered too radical, then adopt the WP changes above, with the addition of:

At the beginning of [lib.locale.cons],

```
  For all constructors except locale(const locale&), any facet found
  in the current global locale at the time of construction, and not
  otherwise specified by arguments, is added to the constructed
  locale.
```

This eliminates the objected-to "caching" behavior, but retains a safe measure of a convenient substitute for transparency, at no objectionable cost in complexity.

3. Adopt the Minimum Reasonable Change, Regardless of Usefulness

Retain locale::transparent() and all its special cases and restrictions on its use. Add the paragraph from (2), replace

the Throws: portion of the description of use_facet<>() as
follows:

```
  Throws: bad_cast if has_facet<Facet>(*this) is false.  Throws other
    unspecified objects catchable as exception on failure.
```

and strike the Effects: portion.

This has the effect of eliminating the universally objected-to "caching" behavior, without violating the design invariant or introducing otherwise-unnecessary changes.

* * *

The three alternatives above eliminate the objected-to "caching behavior" while retaining the necessary object semantics of locales.