COMMENTS ON LOCALE OBJECTS
X3J16/96-0037
WG21/N0855

P.J. Plauger

[At Tom Plum's suggestion, I reproduce here several postings I made to the ad hoc locales reflector Tom established after Tokyo.]

I think it's time to revisit the business of caching locale facets, given the recent discussion of codecvt facets in particular. Here is a brief summary of the relevant issues.

Locales are used widely throughout iostreams, to encapsulate culture-specific information. In the vast majority of uses, a function calls use_facet(getloc(), Facet) to obtain a const reference to a facet of type Facet. The template function use_facet looks first in the locale object delivered up by getloc() (the object imbued into the stream). Only if it can't find such a facet there does it then look in the global locale (the object returned by the default constructor locale()). If neither object contains the facet, use_facet throws bad_cast. Otherwise, use_facet returns the reference, the calling function uses it as necessary, then it discards the reference when its work is done.

References into an object, const or otherwise, can be perilous. The user wants assurances that the reference will not be discredited by a change to the object while the reference is active. The current draft attempts to ameliorate this peril by adopting a simple caching rule -- if use_facet ever successfully returns a facet for a given locale object, it will henceforth always succeed for a similar call on that object. To enforce this invariant:

- The programmer has no way to change the facets that constitute a locale object, once it has been constructed.

- use_facet *does* change the object as needed, by copying a facet found in the global object into the locale object first inspected on a use_facet call.

While this invariant has a certain surface appeal, I maintain that it is neither necessary nor sufficient to provide for safety of references. Moreover, it leads to several forms of undesirable behavior. That's why I have repeatedly argued to eliminate the caching requirement from use_facet.

The invariant is simply not necessary in the vast majority of cases. An iostreams member function calls use_facet, uses the result, then returns. Nowhere in this sequence does an opportunity exist for the program to change either the imbued or the global locale objects. In fact, the only place in the entire library where an opportunity exists is in basic_filebuf. At some point, the object must fixate upon two facets, codecvt<char, traits::char_type, traits::state_type> and codecvt<traits::char_type, char, traits::state_type>, which it uses for subsequent conversions between the internal sequence of traits::char_type elements and the external sequence of char elements. As control passes in and out of the basic_filebuf member functions, ample opportunity exists for the program to change both the imbued locale and the global locale.

The current draft attempts to deal with changes in the imbued locale. Every call to imbue for the stream calls pubimbue for the basic_streambuf, which calls the virtual imbue for the basic_streambuf. Presumably, basic_filebuf supplies its own definition of imbue, which can check for a change of codecvt facets and act accordingly. As we have discussed earlier, however:

- It is easy to mistake an apparently innocuous call to imbue (from the programmer's perspective) for a demand that the basic_filebuf abandon one codecvt discipline for another, with potentially surprising and disastrous results.

- It is not possible, in the general case, to reliably switch codecvt disciplines in mid stream.

That's why I used the term ''fixate'' earlier. My experience is that the only safe way to write basic_filebuf is to have it determine its codecvt discipline early on, then stick with it. It is an easy matter to construct a locale object, contained within basic_filebuf, that assimilates the two codecvt facets for the life of the basic_filebuf object. Once this assimilation is performed, both the locale object imbued in the stream and the global locale are free to change, with no fear that the codecvt references are compromised.

Put simply, caching is not necessary for most calls to use_facet within iostreams. Where it is arguably necessary, for one style of ensuring the reliability of facet references, it is not sufficient. Worse, caching has undesirable behavior:

- Just when a facet is cached depends critically on the pattern of calls to use_facet. A program that changes the global locale object can be surprised to find that a newly introduced facet is ignored by some streams but embraced by others.

- Programs that want to deal in transparent, or semi-transparent locales, are actively thwarted. In particular, the EXISTING PRACTICE of having streams affected by a changing global locale cannot be reproduced.

It is worth repeating that caching never occurs for the commonest uses of the library. The current draft makes it impossible for a conforming program to imbue a locale into any of the predefined streams that lacks any facets used by the library. To set up a situation where caching might make a difference, a program would have to:

- specialize a stream on some element type other than char or wchar_t

- explicitly specialize the two relevant flavors of codecvt

- create a locale object that contains these two facets and make it the global locale

Having done this more than once, I can affirm that it takes considerable sophistication to pull off successfully.

But even if we make transparent locales more widely available, which is also my earnest desire, I see no problems introduced by eliminating caching from use_facet. Quite the contrary, the library becomes simpler and more usable. The caching requirement for use_facet should be struck.

In earlier postings, I've discussed why I think caching should be removed from use_facet. I concur, however, that caching of facets is still at times desirable. What follows is a list of such situations, and how caching can be properly tailored to the job at hand.

First let me repeat that caching is not strictly necessary for the vast majority of calls to use_facet within iostreams, which is the principal client for locale facets within the draft Standard C++ library. Between the use_facet call and the end of the scope that uses the facet reference, no calls can intervene that might invalidate the facet reference. Since caching within the locale object itself leads to surprising and undesirable semantics, it is better to eliminate caching within use_facet completely.

The only place within the current draft where facet references should endure, as I have also observed, is the two codecvt facets used by each basic_filebuf to perform I/O conversions. In this case, you can capture each facet reference in a locale object private to each basic_filebuf object. Thus, given member objects of the form:

```
Infacet *inptr;
Outfacet *output;
locale myloc;
```

you can stabilize the facet references by writing:

```
inptr = (Infacet *)&use_facet<Infacet>(getloc());
outptr = (Outfacet *)&use_facet<Outfacet>(getloc());
myloc = locale(locale(getloc(), inptr), outptr);
```

Subsequent changes to either the global or the imbued locale cannot invalidate the facets pointed to by inptr and output.

So much for the strict requirements. In practice, other considerations intrude. One is a desire for good performance. The need to consult multiple facets while performing I/O leads to some inevitable and unfortunate loss of performance over well tuned C-style I/O. That makes it all the more desirable to eliminate inefficiencies where possible.

One prospect often mentioned is the caching of locale references within an iostream object between calls. The idea is to at least eliminate most calls to use_facet during I/O processing. So within ipfx, for example, you might replace:

```
const ctype<charT>& fac = use_facet<ctype<charT> >(getloc());
```

with:

```
if (ctyptr == 0)
      ctyptr = (ctype<charT> *)&use_facet<ctype<charT> >(getloc());
```

Subsequent references to fac then get replaced by *ctyptr. You have to set the member ctyptr to a null pointer at construction time and on each call to imbue. The payoff is that you typically eliminate all but one call to use_facet.

This particular code relies on the caching of the facet ctype<charT>, within the imbued locale, on the first call to use_facet. (It is currently impossible to cause such caching behavior to actually occur for ctype<char> or ctype<wchar_t>, but the situation *can* arise, even with the status quo.) Only a call to imbue can endanger the validity of ctyptr -- thus the need to clear the pointer within imbue. But what if we remove that guarantee? The code above can change to:

```
if (ctyptr != 0)
      facptr = ctyptr;
else
      {facptr = (ctype<charT> *)&use_facet<ctype<charT> >(getloc());
      if (has_facet<ctype<charT> >(getloc()))
              ctyptr = facptr; }
```

Subsequent references to fac then get replaced by *facptr. This code adapts just as readily to the very common case of a facet in the imbued locale, with essentially the same performance improvement. If the program imbues a locale transparent to ctype<charT>, however, it picks up the facet from the current global locale on each call -- the desired behavior. I maintain that this code is not materially harder to write than the earlier version, yet it supports several desirable ways to use locales in a program.

Two quick comments in passing. I acknowledge that the first example can be made slightly faster. You can put all calls to use_facet in constructors and imbue, thereby saving the test for a null pointer. I also must observe that use_facet can inline quite efficiently. It is questionable, to me at least, whether this form of caching is even worth the bother. Nevertheless, programmers should be aware of their implementation options.

Caching of facet references can be considered important for a slightly different performance reason. Many implementations these days must worry about multi-threading environments. Within the library, that means you have to identify and protect critical sections -- execution intervals over which it is inadvisable to let another thread assume control. While such matters are beyond the scope of the Standard, we certainly don't want to introduce gratuitous impediments within the draft to decent implementation of thread safety in library code.

Obtaining a reference into an object clearly introduces such a critical section. From the time you get such a reference to the time you dispose of it, you don't want to permit another thread to discredit the reference. The current scheme of caching within use_facet minimizes the dangers of changing the global locale object. It reduces the number of critical sections in library code, or at least shortens some of them. But it doesn't solve the whole problem of dangling references. The library must still worry about calls to imbue, within another thread, discrediting a reference in the current thread.

(As an important aside, it is clearly difficult to get sensible behavior in a program that has two threads pounding away at the same stream object. About all a library can typically do is ensure that the behavior stays sane, if not always sensible. I'd rather not go off on a tangent about whether a program *should* be doing some of the things we must nevertheless protect against.)

This is why, in Tokyo, I characterized caching as a performance issue in a multi-threading environment. Any time you implement a library for such an environment, you have to design in critical sections. The conservative approach is to block thread switching freely and often. But such conservatism often leads to lockouts that are both unnecessary and undesirable. The result can be the loss of significant processing overlap, for which the threads were introduced in the first place. Given that an implementation always has to do *something*, most design discussions devolve into arguments about the relative performance costs.

Various schemes exist for minimizing the number, length, and impact of critical sections. That is an essay unto itself. Suffice it to say here that protecting individual facet references is often more desirable than locking out thread switching during great stretches of I/O. And the machinery for doing so already exists, by and large. The essential trick is to create an auto object, private to a thread, which ensures that a reference remains stable while it is being used.

Say you want to be sure that the reference to facet ctype<charT> remains stable during execution of ipfx, but you don't want to prevent all task switching. A simple variant of the trick I showed earlier for basic_filebuf can do the job. Within a critical section you write:

```
locale loc(getloc(),
      (ctype<charT> *)&use_facet<ctype<charT> >(getloc());
```

The facet then survives as long as the dynamic object loc does.

Unfortunately, creating a locale object this way is not very cheap. It may be acceptable each time you construct a new basic_filebuf, but not each time you then read from the stream. I find a simple extension more appealing. Have a variant of use_facet return a smart pointer to the desired facet. The smart pointer asserts ownership when it is constructed by upping the ref count stored in the facet. It lets go by downing the ref count when it is destroyed. While alive, the smart pointer grants ready access to the facet pointer by overloading operator* (and operator->). Critical sections are largely confined to the revised use_facet and the member functions of the smart pointer.

We could add this machinery to the draft, but I see no compelling need to do so. It is the sort of thing you concoct when adding code to handle multi-thread support. No need to standardize *everything*.

In summary, I believe that the current draft offers numerous -- and better -- alternatives to caching facets within use_facet. It is easy to fear the unknown dangers of a change in specification from a paper design. Once you've written the code, however, the dangers can be seen to lie more clearly in the original design itself. Luckily, it is not that hard to debug.

Nathan Myers writes:

> > I've done a bit more analysis, having written any number of
> > code converters over the years, and more than a few codecvt
> > facets this past year. The problem extends beyond locking
> > shift states, because a code converter often stores the *parse*
> > state as well as the shift state in its mbstate_t (or equivalent)
> > object.
>
> I assume that by "parse state" Bill means swallowing the first few
> bytes of a multibyte character. Clearly, if one changes the codeset
> after these bytes have been swallowed, then they must be coughed
> back up, and reinterpreted under the new codeset. This is no
> harder than doing seeks, however.

Thereby reducing to an already unsolved problem? The draft gives no hint about the meaning of streampos objects obtained from a stream that has been translated with multiple codecvt facets. Even assuming we solve this problem (and it isn't an easy one to solve), we must notice that there are seeks and there are seeks. Specifically, the only absolutely portable, reliable seeks are back to a place you've already visited and memorized in a streampos object. Relative seeks aren't guaranteed to work. Nor is more than one character of pushback.

Why does this matter? Because popular coding rules exist that allow an indefinite number of shift codes at the beginning of a character. Some allow multiple ways to encode the same character. So the codecvt facet can't, in general, just look at its remembered parse state and deduce the character sequence that got it there. If it tried to save the raw characters, in a basic_string object for example, it might have to squirrel away a sequence of unbounded length. Even if it can determine what characters to ''cough back,'' it can't depend on pushback to do the job.

The next best prospect is to count the number of characters consumed since the last delivered character, but that is a useful value only for streams that support relative seeks. People fortunate enough to work only under UNIX, and a few systems heavily influenced thereby, will see this as no problem. Those of us who have dealt with the larger world of C implementations for the past quarter century know better. We want to define the C++ Standard, and supply working implementations, that a broad range of customers will find satisfactory.

The only thing I know that ''works'' -- in the sense that it meets the semantic requirements -- is to have basic_filebuf::uflow effectively perform an ftell at the beginning of each character parse. The vast majority of these calls are wasted, but you have to have done one for the most recent character to successfully pull off an arbitrary change of codecvt facets in mid stream. Much nonsense is written about performance, but it is well documented that any file I/O operation that must occur on a per-character basis is going to cause a significant performance hit for many programs. I maintain that this is a high price to pay, for *all* programs, to satisfy the needs of an esoteric few who indulge in mid-stream facet switching.

> In analyzing this issue, we should be careful not to describe
> things as ridiculously difficult if they are equivalent to
> operations we already do, such as seeking.

We should be equally careful not to view the world through rose-colored glasses.

> > I question whether all this machinery is worth adding, to
> > perform an operation that is esoteric at best. Better to
> > leave codecvt switching in mid stream an undefined operation.
>
> Let's not beg the question. Better for whom?

Obviously, a simpler mechanism is better for implementors, because they have less work to do. It is also better for users, absent any other factors, because they will likely get code that has fewer bugs. But that is just one of my concerns. I want semantics simple and obvious enough that producers and consumers will understand it, and not inadvertently proliferate incompatible dialects because of varying interpretations. And I want semantics that can be implemented efficiently and portably in any environment that supports Standard C. Otherwise, we will find our clever standard ignored by significant chunks of the user community.

> If implementors "get it right", nobody else will have to worry

> about it. If implementors welch, then everybody else will have to
> worry. The Japanese, in particular, will be wrestling with codeset
> conversion for a long time, and they are counting on us to get it
> right and not leave them in the cold.

Nothing I have described so far can be construed as leaving the
Japanese ‘‘in the cold.’’ I probably supplied the first C compiler
with Kanji support, almost fifteen years ago. I continue to assist
Japanese companies with both C and C++ compiler and library needs.
My goal is to supply them with something that demonstrably works,
preferably conforming to the C++ Standard. If the C++ Standard
is overly ambitious, however, then it becomes less relevant to my
customers. (They pointedly told me so, again, just a few weeks ago
in Tokyo.)

> > >   BTW, if you’re really intent on reading streams that
> > > identify their encoding in a prefix, you should probably supply a
> > > special codecvt that reads and adapts. (It can also have added member
> > > functions that the program can call to report a safe switch of
> > > encodings.)
>
> Embedding codeset tags invisibly in a stream is a very specialized
> way to communicate. One can hardly pretend that it is general enough
> for uses where codeset information comes from many places, and
> where coeecvt<> implementations are prorietary.

I wasn’t describing embedded codeset tags as a general solution. Quite
the contrary, I chose it as one of an open-ended set of very specific,
and controlled, ways that someone might supply a *well defined* way to
switch code conversions in mid stream. Other *well defined* ways might
require calls to the added member functions of a bespoke codecvt facet,
as I suggested above. Even proprietary codecvt facets can come with a
set of guidelines for when it is safe to switch to and from their
control. Leaving such switching undefined leaves open the possibility
that such a mini market can develop, without saddling the much larger
community with the real and obvious costs of supporting this very
specialized need.

> If identifying problems and fixes necessarily means "larding up",
> then we might as well all go home. The purpose of this discussion
> is to determine how far we can go in supporting users who necessarily
> work with multiple codesets. If you’ve never coded for Asian
> markets, this may look frivolous to you; but it helps nobody to
> keep saying so.

I thought the purpose of this discussion was to determine how to
respond to public comments on the existing draft C++ Standard. Those
comments have identified a number of deficiencies -- many obviously
stemming from the fact that designs have been adopted into the draft
before being implemented. It is now our job to debug these current
designs, not elaborate them even further.

> The question was: is a member in codecvt<> that indicates whether
> the codeset has locking shift states sufficient to determine whether
> changing codesets can be done safely?

And the answer is no.

>   This member would be equivalent
> to mblen(0,0) in the C library. A meaningful answer would detail a
> case where the change could not be made safely, even in the absence
> of locking shift states.

See above for meaningful answer.