ISO/IEC JTC 1/SC 22/WG 21 N4412
Jens Maurer <Jens.Maurer@gmx.net>
2015-04-09

N4412: Shortcomings of iostreams

This paper collects the edited notes from the 2015-02-24 evening session on iostreams held during the LWG meeting in Cologne. I would like to thank all participants, and Dietmar Kühl in particular, for their input.

Use cases

There are a number of communications protocol frameworks in use that employ text-based representations of data, for example XML and JSON. The text is machine-generated and machine-read and should not depend on or consider the locales at either end.

Low-level facilities

Low-level, locale-independent conversion functions for integer/string and floating-point/string conversions should be exposed. Currently, they are assumed to be present at the core of `printf` and `std::num_put` / `std::num_get` (22.4.2 category.numeric), but are not availabe to be called directly from user code. Examples for such functions are `ecvt`, `fcvt`, and `gcvt`.

Futher, there are currently no functions to reliably round-trip floating-point values between binary representation (with fixed size) and decimal representation (with minimal number of digits used). Some environments such as XML Schema require this. See the following papers:

- "How to Print Floating-Point Numbers Accurately" (Guy L. Steele Jr., Jon L. White), https://lists.nongnu.org/archive/html/gcl-devel/2012-10/pdfkieTlklRzN.pdf
- "Printing Floating-Point Numbers Quickly and Accurately with Integers" (Florian Loitsch), http://www.cs.tufts.edu/~nr/cs257/archive/florian-loitsch/printf.pdf
- "How to Read Floating-Point Numbers Accurately" (William D. Clinger), http://www.cesura17.net/~will/Professional/Research/Papers/howtoread.pdf

Overall, a `std::streambuf` seems to provide the right interface for an I/O buffer of unspecified size. However, that interface does not seem to be adequately minimal for byte stream processing (e.g. Base64 encoding or gzip compression).

Text processing should (optionally) be agnostic to different conventions regarding line endings. The existing iostreams offer "text mode" for that.

The equivalent of a `std::filebuf` should not attempt to perform code conversions. Instead, there should be a filtering streambuf that performs code conversions based on the codecvt facilities.

Details for operating system errors (e.g. POSIX errno) leading to I/O failure should be exposed, not concealed.

iostreams interface

Formatting parameters (such as uppercase/lowercase and radix) are specified by setting flags, which mostly persist for an arbitrary number of subsequent low-level formatting operations, until explicitly changed. This approach inhibits compile-time checks and compile-time choice of formatting, and potentially establishes state shared between threads (which requires synchronization for access).

Iostreams are templatized on the character type and the `char_traits`. The latter degree of freedom is rarely exercised, possibly allowing repurposing for an incremental extension of the current iostreams design.

Chaining in the form of "s << a << b << c << d" is a successful interface technique, because it allows to output an arbitrary number of items in a type-safe manner. Overload resolution on operator<< tends to get expensive for larger projects with hundreds or thousands of candidates in the overload set. This seems hard to resolve, since choosing a different name for operator<< simply shifts the expense of overload resolution to a differently-named function.

With C++11, a typesafe "printf"-style interface using variadic templates is possible, also allowing reordering of arguments for output, depending on the format string.

The API of Matt Wilson's FastFormat library at http://sourceforge.net/projects/fastformat/ should be considered, also the various specializations of Boost's `lexical_cast`.

The ability to extend the system to provide input/output primitives for user-defined types is a mandatory feature of a C++ I/O library.

There is currently no use of the money (22.4.6 category.monetary) or time (22.4.5 category.time) formatting facets from the iostreams framework.

It is possible to construct a second i/ostream object using the same `std::streambuf` as an existing i/ostream object. This way, user-defined operator<< functions can quickly obtain an i/ostream object with a well-defined state of the flags.

Internationalization and locales

Locales (see `std::locale`) are integrated at both the `streambuf` and the `iostream` levels. Locale acquisition requires two synchronizations per elementary output call, because locales are a global resource.

Optional internationalization is a mandatory feature of a C++ I/O library, based on the low-level facilities instead of integrated therein.

C++ locales are an incomplete solution for common internationalization requirements; see http://www.boost.org/doc/libs/1_57_0/libs/locale/doc/html/rationale.html#rationale_why. ICU (International Components for Unicode) offer comprehensive internationalization support, albeit with a sub-standard C++ interface. Offering a usable C++ interface based on the ICU feature set (and implementation) would benefit from decades of experience that went into ICU.

For a program, there should be one canonical internal format for internationalized text processing (e.g. UTF-8 or UTF-32), with conversions at the input and output boundaries of the program. An internal representation based on `char32_t` (i.e. UTF-32) has the benefits of a fixed-length encoding, but uses more memory than UTF-8 encoding for most texts.