ISO/IEC JTC1 SC22 WG21 P0067R0
Jens Maurer <Jens.Maurer@gmx.net>
2015-09-25

P0067R0: Elementary string conversions

Introduction

Following up on N4412 "Shortcomings of iostreams", this paper presents low-level, locale-independent functions for conversions between integers and strings and between floating-point numbers and strings.

Use cases include the increasing number of text-based interchange formats such as JSON or XML that do not require internationalization support, but do require high throughput when produced by a server.

There are a lot of existing functions in C++ to perform such conversions, but none offers a high-performance solution. At a minimum, an implementation by an ordinary user of the language using an elementary textbook algorithm should not be able to outperform a quality standard library implementation. The requirements are thus:

- no runtime parsing of format strings
- no dynamic memory allcoation inherently required by the interface
- no consideration of locales
- no indirection through function pointers required
- prevention of buffer overruns
- when parsing a string, errors are distinguishable from valid numbers
- when parsing a string, whitespace or decorations are not silently ignored

For floating-point numbers, there should be a facility to output a floating-point number with a minimum number of decimal digits where input from the digits is guaranteed to reproduce the original floating-point value.

Existing approaches

C++ already provides at least the facilities in the following table, each with shortcomings highlighted in the second column.

facility	shortcomings
sprintf	format string, locale, buffer overrun
snprintf	format string, locale
sscanf	format string, locale
atol	locale, does not signal errors
strtol	locale, ignores whitespace and 0x prefix
strstream	locale, ignores whitespace
stringstream	locale, ignores whitespace, memory allocation
num_put / num_get facets	locale, virtual function
to_string	locale, memory allocation
stoi etc.	locale, memory allocation, ignores whitespace and 0x prefix, exception on error

As a rough performance comparison, the following simple numeric formatting task was implemented: Output the integer numbers 0 ... 1 million, separated by a single space character, into a contiguous array buffer of 10 MB. This task was executed 10 times. The execution environment was gcc 4.9 on Intel Core i5 M450.

strstream	864 ms	uses `std::strstream` with application-provided buffer
streambuf	540 ms	uses simple custom streambuf with `std::num_put<>` facet
direct	285 ms	open-coded "divide by 10" algorithm, using the interface described below
fixed-point	125 ms	fixed-point algorithm found in an older AMD optimization guide, using the interface described below

There are various approaches for even more efficient algorithms; see, for example, https://gist.github.com/anonymous/7700052 .

Interface discussion

The following discussion assumes that a common interface style should be established that covers (built-in) integer and floating-point types. The type `T` designates such an arithmetic type. Note that given these restrictions, output of T to a string has a small maximum length in all cases. The styles for input vs. output will differ due to the differing functionality.

The fundamental interface for a string is that it is caller-allocated, contiguous in memory, and not necessarily 0-terminated. That means, it can be represented by a range [`begin`,`end`) where `begin` and `end` are of type `char *`.

Given this framework, the following subsections discuss various specific interface styles for both output and input. In each case, the signature of an integer output or input function is shown. Criteria for comparison include impact on compiler optimizations, indication of output buffer overflow, and composability (as a measure of ease-of-use).

Output

This subsection discusses various specific interface styles for output. In each case, the signature of an integer output function is shown. There is one failure mode for output: overflow of the provided output buffer. Criteria for comparison include impact on compiler optimizations, indication of output buffer overflow, and composability (as a measure of ease-of-use). For exposition of the latter, consecutive output of two numbers is shown, without any separator.

Conceptually, an output function has four parameters and two results. The parameters are the `begin` and `end` pointers of the buffer, the value, and the desired base. The results are the updated `begin` pointer and an overflow indication.

Iterator

```
char * to_string(char * begin, char * end, T value, int base = 10);
```

This interface style returns the updated `begin` pointer. That is, the resulting string is in [`begin`, *return-value*) and [*return-value*, `end`) is unused space in the string. Such an interface style is used for many standard library algorithms, e.g. `find` [alg.find]. All parameters are passed by value which helps the optimizer. Overflow is indicated by *return-value* == `end`. The situation that the output exactly fits into the provided buffer cannot be distinguished from overflow. Two consecutive outputs can be produced trivially using:

```
p = to_string(p, end, value1);
p = to_string(p, end, value2);
```

Iterator with in-situ update

```
void to_string(char *& begin, char * end, T value, int base = 10);
```

This interface style updates the `begin` pointer in place. That is, the resulting string is in [`old-begin`, `begin`) and [`begin`,`end`) is unused space in the string. Aliasing rules allow that updates to `begin` change the data where begin points. To avoid redundant updates, the implementation can copy `begin` to a local variable. Overflow is indicated by `begin` reaching `end`. The situation that the output exactly fits into the provided buffer cannot be distinguished from overflow. Two consecutive outputs can be produced trivially using:

```
to_string(p, end, value1);
to_string(p, end, value2);
```

string_view

```
void to_string(std::string_view& s, T value, int base = 10);
```

This interface style groups the `begin` and `end` pointers into a `string_view` which is updated in-place. Comments on "iterator with in-situ update" apply analogously.

Iterator with in-situ update and overflow indication

Adding a boolean return value allows to indicate overflow:

```
bool to_string(char *& begin, char * end, T value, int base = 10);
```

Comments on "iterator with in-situ update" apply analogously, except that the return value indicates whether overflow occurred.

snprintf

```
int to_string(char * begin, char * end, T value, int base = 10);
```

This interface style always returns the number of characters required to output T, regardless of whether sufficient space was provided. That is, an overflow occurred if the return value is larger than `end-begin`, otherwise the resulting string is in [`begin`, `begin + `*`return-value`*). Such an interface style is used for `snprintf`, except that the proposed function never 0-terminates the output. All parameters are passed by value which helps the optimizer. Overflow is indicated by a return value strictly larger than the distance between `begin` and `end`. Two consecutive outputs require attention at the caller site to avoid buffer overflow:

```
int n = 0;
n += to_string(begin, end, value1);
```

```
  n += to_string(begin + std::min(n, end-begin), end, value2);
```

Conclusion

For me, the "iterator" approach seems to blend in best with the rest of the standard library. Loss of exact overflow indication seems not too important. If "total result size" is an important information, the "snprintf" interface delivers that plus an exact overflow indication.

Input

An input function conceptually operates in two steps: First, it consumes characters from the input string matching a pattern until the first non-matching character or the end of the string is encountered. Second, the matched characters are translated into a value of type `T`. There are two failure modes: no characters match, or the pattern translates to a value that is not in the range representable by `T`.

Conceptually, an input function has three parameters and three results. The parameters are the `begin` and `end` pointers of the string and the desired base. The results are the updated `begin` pointer, a `std::error_code` and the parsed value.

This subsection discusses various specific interface styles for input. Failure is indicated by `std::error_code` with the appropriate value. In each case, the signature of an integer input function is shown. Criteria for comparison include impact on compiler optimizations and composability (as a measure of ease-of-use). For exposition of the latter, parsing of two consecutive values is shown, without skipping of any separator.

Iterator

```
  const char * from_string(const char * begin, const char * end, T& value, std::error_code& ec, int base = 10);
```

This interface style returns the updated `begin` pointer. That is, the returned pointer points to the first character not matching the pattern. Such an interface style is used for many standard library algorithms. All parameters are passed by value which helps the optimizer. Two consecutive inputs can be performed like this:

```
  T value1, value2;
  std::error_code ec;
  p = from_string(p, end, value1, ec);
  if (ec)
    /* parse error */;
  p = from_string(p, end, value2, ec);
  if (ec)
    /* parse error */;
```

Iterator with in-situ update

```
  void from_string(const char *& begin, const char * end, T& value, std::error_code& ec, int base = 10);
```

This interface style updates the `begin` pointer in place. Two consecutive inputs can be performed like this:

```
  T value1, value2;
  std::error_code ec;
  from_string(p, end, value1, ec);
  if (ec)
    /* parse error */;
  from_string(p, end, value2, ec);
  if (ec)
    /* parse error */;
```

Iterator with in-situ update and error return

```
  std::error_code from_string(const char *& begin, const char * end, T& value, int base = 10);
```

Returning the error code allows for more compact code at the call site:

```
  T value1, value2;
  if (std::error_code ec = from_string(p, end, value1))
    /* parse error */;
  if (std::error_code ec = from_string(p, end, value2))
    /* parse error */;
```

Return a std::pair or std::tuple

Two of the three results of an input function could be represented by a pair. All three results could be represented by a tuple. However, experience with `std::map` shows that the naming of the parts (`first` and `second`) carries no semantic meaning which would help reading the resulting code. If the result value moves to the return value, its type `T` needs to be passed explicitly (e.g. as a template parameter). The composition example would be:

```
  std::pair<T, std::error_code> res;
  res = from_string<T>(p, end);
  if (res.second)
    /* parse error */;
  T value1 = res.first;
  res = from_string<T>(p, end);
  if (res.second)
    /* parse error */;
  T value2 = res.second;
```

Conclusion

The "iterator" style seems to blend in best with the rest of the standard library. The "iterator with in-situ update and error return" allows for the most compact code at the call site. Returning a pair or tuple seems least attractive.

Naming

This paper proposes `to_string` (overloaded from its existing usage in the standard library) for the output function and `from_string` for the input (parse) function. I'm open for other suggestions.

Details

Output

An *output function* is a function with the declaration pattern

```
char * to_string(char * begin, char * end, T value /* possibly more parameters */);
```

for some type T, converting `value` into a character string by successively filling the range [`begin`,`end`) and returning the one-past-the-end pointer where the conversion was complete. [Note: The resulting string is not null-terminated.] If the provided space is insufficiently large, returns `end`; the contents of the range [`begin`,`end`) are unspecified.

For each integer type (including extended integer types) `T` except `char` and `bool`, the following output function is provided:

```
char * to_string(char * begin, char * end, T value, int base = 10);
```

Valid values for `base` are between 2 and 36 (inclusive). Digits in the range 10..36 are represented as lowercase characters a..z. The value in `value` is converted to a string of digits in the given `base` (with no redundant leading zeroes). If T is a signed integral type, the representation starts with a minus sign if `value` is negative.

For each floating-point type `T`, the following output function is provided:

```
char * to_string(char * begin, char * end, T value);
```

A finite `value` is converted to a leading minus sign if `value` is negative, a string of decimal digits with at most one decimal point, and an optional trailing exponent introduced by a lowercase "e" followed by an optional minus sign followed by decimal digits. The representation requires a minimal number of characters, yet parsing the representation using the `from_string` function recovers `value` exactly. If `value` is an infinity, it is converted to `inf` or `-inf`; a NaN is converted to `nan` or `-nan`.

For each floating-point type `T`, the following output function is provided:

```
char * to_string(char * begin, char * end, T value, int precision);
```

A finite `value` is converted to a leading minus sign if `value` is negative and a string of decimal digits with at most one decimal point and exactly `precision` digits after the decimal point. If `precision` is 0, no decimal point appears, otherwise at least one digit appears before the decimal point. The result is correctly rounded, i.e. it yields the same result as an infinite-precision output rounded to the desired precision.

Input

An *input function* is a function with the declaration pattern

```
const char * from_string(const char * begin, const char * end, T& value, std::error_code& ec /* possibly more parameters */);
```

for some type T. An input function analyzes the string [`begin`, `end`) for a pattern. If no characters match the pattern, the function retuns `begin`, `value` is unmodified, and `ec` is set to `EINVAL`. Otherwise, the characters matching the pattern are interpreted as a representation of a value of type T. The function returns a pointer that points to the first character not matching the pattern. If the parsed value is not in the range representable by T, `value` is unmodified and `ec` is set to `ERANGE`. Otherwise, `ec` is set such that conversion to `bool` yields false.

For each integer type (including extended integer types) `T` except `char` and `bool`, the following input function is provided:

```
const char * from_string(const char * begin, const char * end, T& value, std::error_code& ec, int base = 10);
```

Valid values for base are between 2 and 36 (inclusive). Digits in the range 10..36 are represented as lowercase characters a..z. If `T` is an unsigned integral type, the pattern is a non-empty sequence of digit characters according to `base`. If `T` is a signed integral type, the pattern is an optional minus sign followed by a non-empty sequence of digit characters.

For each floating-point type `T`, the following input function is provided:

```
const char * from_string(const char * begin, const char * end, T& value, std::error_code& ec);
```

The pattern is an optional minus sign followed by a non-empty sequence of decimal digit characters with at most one decimal point, and an optional trailing exponent introduced by "`e`" or "`E`" followed by an optional minus sign followed by a non-empty sequence of decimal digits. The resulting `value` is one of at most two floating-point values closest to the given pattern. The pattern also matches the representations of infinity and NaN described for output.

Open Issues

- review input/output pattern specifications vs. C standard "fprintf"
- discuss naming

References

- "How to print floating-point numbers accurately" by Guy L. Steele Jr. and Jon L White, Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation; http://www.kurtstephens.com/files/p372-steele.pdf (starting at page 4 of the PDF)
- "Printing Floating-Point Numbers Quickly and Accurately with Integers" by Florian Loitsch, PLDI'10 http://www.cs.tufts.edu/~nr/cs257/archive/florian-loitsch/printf.pdf
- "How to Read Floating Point Numbers Accurately" by William D Clinger, University of Oregon http://www.cesura17.net/~will/professional/research/papers/howtoread.pdf