P3374:Adding formatter for fpos<mbstate_t>

Liang Jiaming

Motivation

- The first draft
  - Thanks to Victor and Arthur!

Before

```
std::stringstream s{"ABC"};
std::cout << s.tellg();        // ✅Yes on almost all implementations
std::println("{}", s.tellg()); // ❌Compile error...
// To make it work, we have to write...
std::println("{}", (std::streamoff)s.tellg());    // 😳What?
std::println("{}", s.tellg() - std::streampos{}); // 😳No way...
```

After

```
std::stringstream s{"ABC"};
std::cout << s.tellg();        // ✅Yes on almost all implementations
std::println("{}", s.tellg()); // ✅Yes!
```

- Key problem at that time: how to implement it?
  - By specialization, as we do in proxy type for e.g. `std::vector<bool>`;
  - By ADL-based methods, similar to P3070 by Victor (which is for enumerations but could be extended to any types). This would lead to smaller binary code size.

Motivation

- When I sent the draft to SG16 mailing list, I got feedback from Tom immediately (and fortunately).
  - Thanks to Tom!
- I forgot the template parameter, the state type 😢 .
  - And the output by stream is not watertight since the state is always neglected.
- Problem now: how to design a format specification for integer + state.
  - Current proposal: fully leverage the format specification for integers.

Before

```
std::ofstream s{"some_file"};
// Do some output...
std::cout << s.tellg();        // ❓Yes on almost all implementations, but not robust
std::println("{}", s.tellg()); // ❌Compile error...
// To make it work same as operator<<, we have to write...
std::println("{}", (std::streamoff)s.tellg());    // 😳What?
std::println("{}", s.tellg() - std::streampos{}); // 😳No way...
```

After

```
std::ofstream s{"some_file"};
// Do some output...
std::cout << s.tellg();          // ❓Yes on almost all implementations, but not robust
std::println("{}", s.tellg());    // ✅Yes and robust!
std::println("{:d}", s.tellg());  // ✅Non-robust way can be controlled by users explicitly.
std::stringstream s2{"ABC"};
std::println("{:d}", s2.tellg()); // ✅Especially for streams that don't need codecvt.
```

Reason

- Why is the state type important?
  - Encoding conversion.
  - For example, “水” has 2 bytes (1 code unit) in UTF-16 while 3 bytes (3 code units) in UTF-8.
    - If we want to convert UTF-16 to `char8_t[2]`, then it’s incomplete.
    - We need to record where we are to continue the parsing, since the intermediate state is not a valid Unicode code point.
  - To abstract the intermediate state for different encodings, C++ uses `mbstate_t`.
    - And to ensure that some unknown encodings can also be converted to/from, both `codecvt` and `fpos` have a state type template parameter.

Reason

- Why is it always neglected?
  1. Though the standard only regulates it should be convertible to integers **by explicit cast** (i.e. `(streamoff)pos`), all standard library implementations make the conversion operator **non-explicit**.
     - Thanks to Jens, it's due to absence of explicit conversion operator in C++98; and implementations don't do any enhancement though C++11 introduces that.
     - And that's why we could output the position by stream directly, since it matches one of the integer overloads.
  2. To maintain such illusion, current implementations try to ensure the state type can be neglected safely.
     - The conversion is always complete for `fstream` in implementations.

```
if (this->pbase() < this->pptr())
  {
    // If appropriate, append the overflow char.
    if (!__testeof)
      {
        *this->pptr() = traits_type::to_char_type(__c);
        this->pbump(1);
      }

    // Convert pending sequence to external representation,
    // and output.
    if (_M_convert_to_external(this->pbase(),
                               this->pptr() - this->pbase()))
      {
        _M_set_buffer(0);
        __ret = traits_type::not_eof(__c);
      }
  }
```

libstdc++ policy: allocate the maximum buffer.

Overflow in **`filebuf`** → **`codecvt`** to intermediate buffer → output to the file.

```
template<typename _CharT, typename _Traits>
  bool
  basic_filebuf<_CharT, _Traits>::
  _M_convert_to_external(_CharT* __ibuf, streamsize __ilen)
  {
    // Sizes of external and pending output.
    streamsize __elen;
    streamsize __plen;
    if (__check_facet(_M_codecvt).always_noconv())
      {
        __elen = _M_file.xsputn(reinterpret_cast<char*>(__ibuf), __ilen);
        __plen = __ilen;
      }
    else
      {
        // Worst-case number of external bytes needed.
        // XXX Not done encoding() == -1.
        streamsize __blen = __ilen * _M_codecvt->max_length();
        char* __buf = static_cast<char*>(__builtin_alloca(__blen));

        char* __bend;
        const char_type* __iend;
        codecvt_base::result __r;
        __r = _M_codecvt->out(_M_state_cur, __ibuf, __ibuf + __ilen,
                              __iend, __buf, __buf + __blen, __bend);

        if (__r == codecvt_base::ok || __r == codecvt_base::partial)
          __blen = __bend - __buf;
        else if (__r == codecvt_base::noconv)
          {
            // Same as the always_noconv case above.
            __buf = reinterpret_cast<char*>(__ibuf);
            __blen = __ilen;
          }
        else
          __throw_ios_failure(__N("basic_filebuf::_M_convert_to_external "
                                  "conversion error"));
        __elen = _M_file.xsputn(__buf, __blen);
        __plen = __blen;

        // Try once more for partial conversions.
        if (__r == codecvt_base::partial && __elen == __plen)
          {
            const char_type* __iresume = __iend;
            streamsize __rlen = this->pptr() - __iend;
            __r = _M_codecvt->out(_M_state_cur, __iresume,
                                  __iresume + __rlen, __iend, __buf,
                                  __buf + __blen, __bend);
            if (__r != codecvt_base::error)
              {
                __rlen = __bend - __buf;
                __elen = _M_file.xsputn(__buf, __rlen);
                __plen = __rlen;
              }
            else
              __throw_ios_failure(__N("basic_filebuf::_M_convert_to_external "
                                      "conversion error"));
          }
      }
```

If **`.encoding() == -1`**, then **`.max_length()`** can be fake.

libstdc++ bug: only try once more and always assume it's complete.

```
if (this->pptr() != this->pbase()) {
  if (__always_noconv_) {
    size_t __nmemb = static_cast<size_t>(this->pptr() - this->pbase());
    if (std::fwrite(this->pbase(), sizeof(char_type), __nmemb, __file_) != __nmemb)
      return traits_type::eof();
  } else {
    char* __extbe = __extbuf_;
    codecvt_base::result __r;
    do {
      if (!__cv_)
        __throw_bad_cast();

      const char_type* __e;
      __r = __cv_->out(__st_, this->pbase(), this->pptr(), __e, __extbuf_, __extbuf_ + __ebs_, __extbe);
      if (__e == this->pbase())
        return traits_type::eof();
      if (__r == codecvt_base::noconv) {
        size_t __nmemb = static_cast<size_t>(this->pptr() - this->pbase());
        if (std::fwrite(this->pbase(), 1, __nmemb, __file_) != __nmemb)
          return traits_type::eof();
      } else if (__r == codecvt_base::ok || __r == codecvt_base::partial) {
        size_t __nmemb = static_cast<size_t>(__extbe - __extbuf_);
        if (fwrite(__extbuf_, 1, __nmemb, __file_) != __nmemb)
          return traits_type::eof();
        if (__r == codecvt_base::partial) {
          this->setp(const_cast<char_type*>(__e), this->pptr());
          this->__pbump(this->epptr() - this->pbase());
        }
      } else
        return traits_type::eof();
    } while (__r == codecvt_base::partial);
  }
```

.overflow

libc++ policy: use a fixed-length buffer and loop until complete.

```
template <class _CharT, class _Traits>
basic_filebuf<_CharT, _Traits>::basic_filebuf()
    : __extbuf_(nullptr),
      __extbufnext_(nullptr),
      __extbufend_(nullptr),
      __ebs_(0),
      __intbuf_(nullptr),
      __ibs_(0),
      __file_(nullptr),
      __cv_(nullptr),
      __st_(),
      __st_last_(),
      __om_(0),
      __cm_(__no_io_operations),
      __owns_eb_(false),
      __owns_ib_(false),
      __always_noconv_(false) {
  if (std::has_facet<codecvt<char_type, char, state_type> >(this->getloc())) {
    __cv_            = &std::use_facet<codecvt<char_type, char, state_type> >(this->getloc());
    __always_noconv_ = __cv_->always_noconv();
  }
  setbuf(nullptr, 4096);
}
```

```
if (__ebs_ > sizeof(__extbuf_min_)) {
  if (__always_noconv_ && __s) {
    __extbuf_  = (char*)__s;
    __owns_eb_ = false;
  } else {
    __extbuf_  = new char[__ebs_];
    __owns_eb_ = true;
  }
```

.setbuf

```
int_type __CLR_OR_THIS_CALL overflow(int_type _Meta = _Traits::eof()) override { // put an element to stream
    if (_Traits::eq_int_type(_Traits::eof(), _Meta)) {
        return _Traits::not_eof(_Meta); // EOF, return success code
    }

    if (_Mysb::pptr() && _Mysb::pptr() < _Mysb::epptr()) { // room in buffer, store it
        *_Mysb::_Pninc() = _Traits::to_char_type(_Meta);
        return _Meta;
    }

    if (!_Myfile) {
        return _Traits::eof(); // no open C stream, fail
    }

    _Reset_back(); // revert from _Mychar buffer
    if (!_Pcvt) { // no codecvt facet, put as is
        return _Fputc(_Traits::to_char_type(_Meta), _Myfile) ? _Meta : _Traits::eof();
    }

    // put using codecvt facet
    constexpr size_t _Codecvt_temp_buf = 32;
    char _Str[_Codecvt_temp_buf];
    const _Elem _Ch = _Traits::to_char_type(_Meta);
    const _Elem* _Src;
    char* _Dest;

    // test result of converting one element
    switch (_Pcvt->out(_State, &_Ch, &_Ch + 1, _Src, _Str, _Str + _Codecvt_temp_buf, _Dest)) {
    case codecvt_base::partial:
    case codecvt_base::ok:
        { // converted something, try to put it out
            const auto _Count = static_cast<size_t>(_Dest - _Str);
            if (0 < _Count && _Count != static_cast<size_t>(_CSTD fwrite(_Str, 1, _Count, _Myfile))) {
                return _Traits::eof(); // write failed
            }

            _Wrotesome = true; // write succeeded
            if (_Src != &_Ch) {
                return _Meta; // converted whole element
            }

            return _Traits::eof(); // conversion failed
```

```
void _Init(FILE* _File, _Initfl _Which) noexcept { // initialize to C stream _File after {new, open, close}
    using _State_type = typename _Traits::state_type;

    __PURE_APPDOMAIN_GLOBAL static _State_type _Stinit; // initial state

    _Closef    = _Which == _Openfl;
    _Wrotesome = false;

    _Mysb::_Init(); // initialize stream buffer base object

    if (_File && sizeof(_Elem) == 1) { // point inside C stream with [first, first + count) buffer
        _Elem** _Pb = nullptr;
        _Elem** _Pn = nullptr;
        int* _Nr    = nullptr;

        ::_get_stream_buffer_pointers(
            _File, reinterpret_cast<char***>(&_Pb), reinterpret_cast<char***>(&_Pn), &_Nr);
        int* _Nw = _Nr;

        _Mysb::_Init(_Pb, _Pn, _Nr, _Pb, _Pn, _Nw);
    }

    _Myfile = _File;
    _State  = _Stinit;
    _Pcvt   = nullptr; // pointer to codecvt facet
}
```

Calls e.g. setp here.

Otherwise “unbuffered” from the view of **`basic_streambuf`**, so every write will call **`.overflow`**/**`.xsputn`** (and **`.xsputn`** will also call **`.overflow`**).

MS-STL policy: if **`sizeof(CharT) != 1`**, keep the filebuf “unbuffered” (actually buffered by **`FILE*`**) and always call **`.overflow`** element by element so conversion happens one by one.

Reason

- But, if users define some customized stream buffer & stream so that…
  - They hope it to flush lazily;
  - They think the performance sucks, e.g. for MS-STL, it uses the virtual call `.overflow` over and over again.
  - …
- Then they may need partial conversion, and the state is not neglectable.
- We know that `fpos` is designed to be convertible to & from integer because we hope to somehow operate it like an integer conveniently, including recording and recovering.
  - However, there is never a way for users to dump mbstate_t and recover it later.
- And that's why I write this version of proposal…

Proposal

- Current solution:

  So to be specific, the formatter specialization of `fpos<mbstate_t>` should behave as follows:

  — When no specification is given (like `{}` or `{:}`), `format` should produce `"(position, mbstate descriptor)"`, where `mbstate descriptor` can be used to fully restore the state of `fpos`;

  — When some specifications are given (e.g. `{:d}`), only the position will be output in the way determined by the format specifications.

- Before all other problems, we need to determine:
  - **Is it considered useless by SG16 to support format for fpos?**
  - **If it's not, is it considered useless by SG16 to dump mbstate_t? Is it enough to only format the integer?**

Undetermined Design

1. As suggested by Arthur, do we need to force the conversion operator of `fpos` to be explicit?
   - This is a breaking change.
2. For format specifications without the state, do we need to design some safe specifications to check whether the `mbstate_t` is in the initial state?
   - i.e. it's safe to neglect the state.
3. Do we need to apply some explicit constraint to format of the state?
   - `"(position, state descriptor)"` may be ambiguous for `std::scan` to parse, e.g. `"(1, )"aa)"`.
4. Do we need to add formatter for `mbstate_t` itself?

Thank you!

Special acknowledgement to Arthur, Victor, Tom,
and all guys who helped me before 🥰.