A RUDIMENTARY UNICODE ABSTRACTION

Attempting to wrangle encoding and more

ThePhD – March 3rd, 2018
@thephantomderp
C++ Text Working Group, #std-text-wg for Cpplang Slack

WHY?

- std::wstring_convert is sort of terrible
  - thankfully has been deprecated
- Needed to not sprinkle to_utfX or to_string<utfX> all over API boundaries
  - leaves internal format a mystery until you look at boundary code / docs
  - std::string / std::wstring poor descriptors
    - std::u16string is a name, but it enforces and helps with nothing

SO ANOTHER STRING CLASS?

- Two of them, actually
  - text_view -> basic_text_view<Char, Encoding, ErrorPolicy>
  - text -> basic_text<Storage, Encoding, ErrorPolicy>

THE TIMES

- Clang (3.3/3.4) and GCC (4.9, 5.x)
  - the cool toys I used to do things and then prepared for Stage II: Reality
- Sanity's Eclipse, 2013
  - Also known as Visual Studio
  - November CTP, ICEs on the daily "Hey this compiles in GCC surely it- oh…"
- C++11, barely scratching 14
  - "C++07", with VC++

BASIC_TEXT_VIEW, VERSION 1

- Codepoint abstraction, using template parameters:
  - Char (= char) – pick code unit denomination
  - Encoding (= default_encoding_t<Char>) – controlled what underlying contiguous sequence would be treated as
    - Encoding.decode(Iter, Iter), Encoding.encode(Iter, Iter) [+ reverse]
  - ErrorPolicy (= detail::default_error_policy) – controlled how encoding errors would be handled
    - Policy(Encoding&, encoded/decoded_result, Iter, Iter) – do whatever

NOT… QUITE!

- Since we pass Char template, assumes unit of storage is const char*
  - Contiguous storage got us 90% of the way there, though
  - Honestly good enough™

- Nevertheless, modify interface anyways for completeness's sake
  - basic_string_view<Char> handled viewing a basic string
  - Side note: handled mutability / immutability for us, but std:: is different
    - const Char = immutable | Char = mutable

BASIC_TEXT_VIEW, VERSION 2

- basic_text_view<Rangeable, Encoding, ErrorPolicy>
  - "look at us we're cool with ranges!"
  - boiled down to "listen just get the begin/end and shovel it through algos"
  - could never touch range-v3 with 10 ft. pole on Visual Studio
  - Wrote encoding_/decoding_iterator<BaseIt, Encoding, ErrorPolicy>

- Handled all immutable algorithms, offered codepoint interface
  - find/rfind, starts_with, ends_with, search, compare, etc…

BASIC_TEXT

- basic_text<Storage, Encoding, ErrorPolicy>
  - Similar to last, except template parameter Storage – just what we are storing

- struct basic_text : basic_text_view<Storage, Encoding, ErrorPolicy>
  - No duplication, all member functions get carted over, it's all so nice
  - basic_text_view handles the storage for free
  - insert, erase, append, prepend, replace, to_upper/lower/title

CONVERTING CONSTRUCTORS I

- using text = basic_text<std::string>; // std::string is assumed to be utf8
  - text bark(u"🐶"); // assumes utf16, default_policy
  - text wine(U"🍷"); // assumes utf32, default_policy
  - text wine2(u8"🍷"); // assumes utf8, default_policy
  - text for_char_literals("abcd"); // assumes utf8, default_policy
  - text my_text(U"🍷"); // assumes utf32, default_policy
- basic_text<std::string, ascii> ascii_text(u"🔥"); // static_assert triggers
- basic_text<std::string, ascii> ascii_text(u"☢", utf16{}, my_policy{}); // ok…

CONVERTING CONSTRUCTORS II

- Not sure if best implementation, truly
  - converting constructors can be expensive, not sure I'd want to standardize
  - or if I'd want to do it again, really

- But catching the errors with static_asserts and similar was nice!
  - Having constructors similar to std::string was useful for codebase integration
  - Most understood converting constructors cost, easily used string/text_view when performance mattered

IMPLEMENTATION DETAILS

- Also had “allowed upgrades”
  - e.g. example ascii -> utf8 did not trigger a static assert
    - ☢ BAD: also was optimized to just memcpy bits! ☢
    - ☢ Did not require a policy that allowed for such: could be invalid ASCII ☢
- comparisons used codepoints, optimizations ✔ encoding/storage equivalent
  - applied this internally to everything, since replace/append/prepend could take arbitrary ranges with optional policy/char-range

⚠ PROBLEMATIC ⚠

- “ў” != “ў”
  - ??? What?

- Welcome to the Combining Codepoint Fair!
  - Cyrillic y combined with breve (˘): ў = 2 codepoints
  - Visually identical and canonically equivalent, but Short U (Cryllic): ў = 1 codepoint

- Reversing a string does not split up code units, but combined codepoints split

PREPARE FOR FUTURE

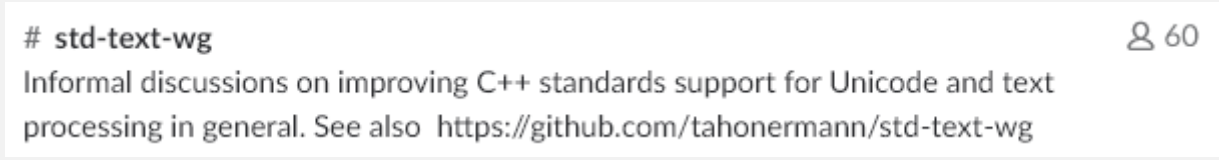
- Time to look into Normalization and Segmentation…
  - Normalization and Segmentation can be done as iterators alone?
  - Locales needed for graphemes / extended grapheme clusters, not normalization

- Prepared UCD (Homerolled)
  - Release-mode 8.5~8.8 megabytes, definitely not best compression
    - after beating it up to get around string limits / initializer_list limits in Visual Studio
  - Included most everything, even Han data

TIME FOR NORMALIZA-

- And then life happened.
  - Was going to pick Normalization Form Compatibility Decomposition + Canonical Composition, NFKC
    - Stable Code Points property mentioned in UAX #15 §9
    - Best for comparing with the outside world, not the best for internal processing?

- 3 or so years later, I enter a Slack Workspace, and there's this channel named…



WHAT TO DO: A CONVERSATION

ThePhD: So tzlaine/text it's like Tom's text_view, but stapled to utf8?

tzlaine: Yes. With quite a few staples.

WHAT TO DO

- De-couple the text class (and the rope class, too!) from its utf8 storage mechanism

- Split asunder:
  - into a text_view alike class similar to what Tom Honermann's is
  - into text class alike to what I used to have (except way better)
  - hammer out free functions and their interfaces (transcoding, etc.)
  - Enjoy the beautiful new C++17/20 in Visual Studio save ICE for hot days 😎

TIME SPLIT

- 1st Priority: figure out how to interact with the committee
  - Do this with less-important std::embed(…) proposal (unrelated)
- Work on de-stapling tzlaine/text from its utf8 representation
- Look at piling more UCD data into a complete database for C++ consumption
  - Perhaps standardized way to query such data? (Maybe as an extension?)