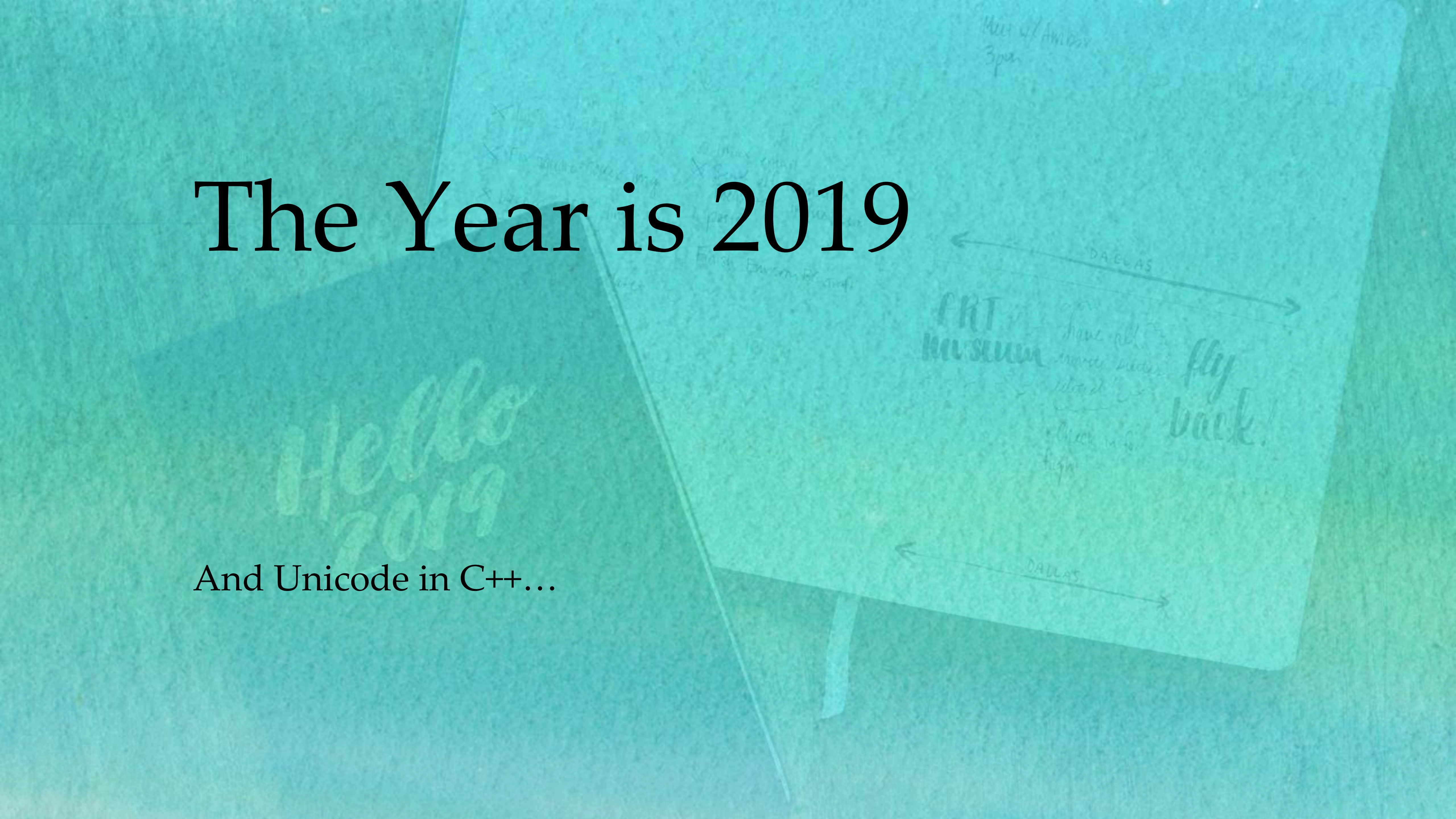
Catching ⍐

The (Baseline) Unicode Plan for C++23

ThePhD | @thephantomderp | phdofthehouse@gmail.com
https://linkedin.com/in/thephd | https://thephd.github.io | https://github.com/ThePhD



The Year is 2019

And Unicode in C++…

… really bites the dust.

- `std::wstring_convert` needed to be pulled
  - Caused great harm to performance;
  - poorly implemented (thanks, `std::locale`!)

- Generally, `codecvt` / facets are terrible
  - Virtual interfaces
  - No data polymorphism or data type extensions

Pain Points

What makes text conversion hard?

Conflating Locale + Encoding ☹

- Save a non-ASCII file in Germany, ship it to Japan
  - Cry bitter tears when the bug reports roll in

- Use /utf8 in MSVC
  - Forget to save your file as UTF8 in the editor, cry bitter tears

- Ship code to platform you don't own
  - Locale is different: wonder why no bug is reproducible

“What is the Encoding?”

- `char`
  - whatever system feels like!
  - LANG, setlocale, Active Code Page, computer location, …

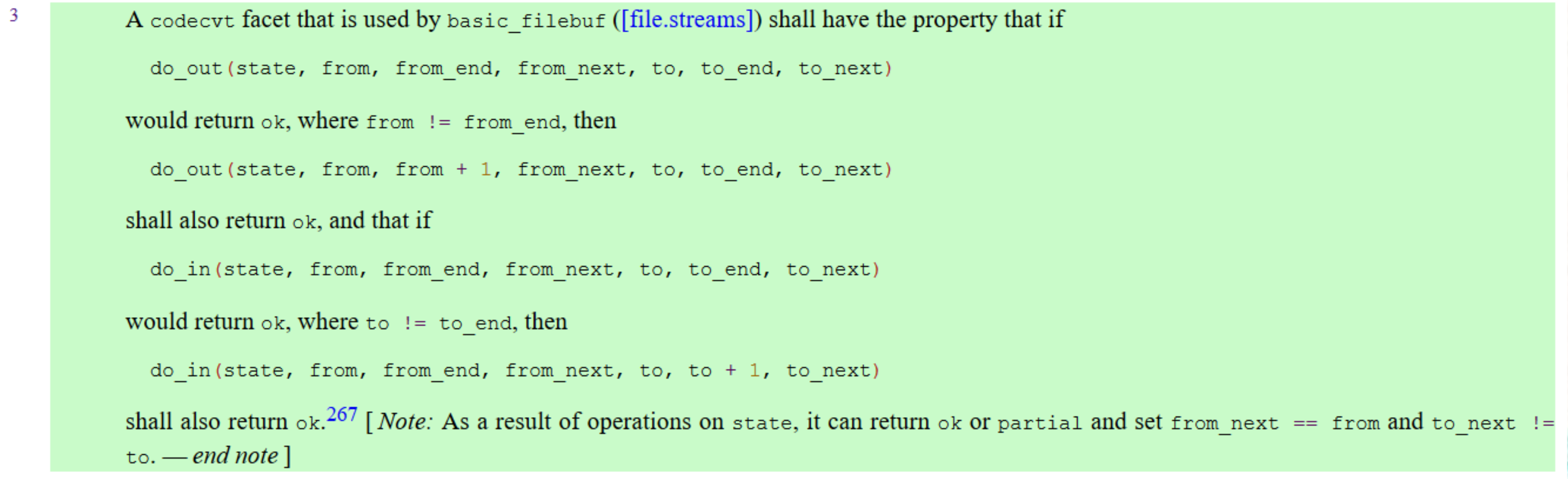
- `wchar_t`
  - UTF16 Windows/IBM, UTF32 not-Windows…
  - … maybe! UTF32 for IBM platforms at 64-bit, UTF16 otherwise.
  - … exception for Chinese locales, where it’s some flavor of Big5 or GBK or something!

“Well we have Unicode Literals!”

- Nope: beholden to macros.
- `char16_t`: `__STD_C_UTF16__`
- `char32_t`: `__STD_C_UTF32__`
- Fixed in C++20 (p1041 – *Make char16/32_t UTF16/32*)
  - thanks, Robot!

“Okay, but C++ streams are—”

- No.
  - wchar_t still horrible even with “multiple code units” allowed in streams!
  - basic_filebuf is restricted to 1:1 conversion for all in/output code units [locale.codecvt.virtuals#3]



3 A codecvt facet that is used by basic_filebuf ([file.streams]) shall have the property that if

```
do_out(state, from, from_end, from_next, to, to_end, to_next)
```

would return ok, where from != from_end, then

```
do_out(state, from, from + 1, from_next, to, to_end, to_next)
```

shall also return ok, and that if

```
do_in(state, from, from_end, from_next, to, to_end, to_next)
```

would return ok, where to != to_end, then

```
do_in(state, from, from_end, from_next, to, to + 1, to_next)
```

shall also return ok.[267] [*Note:* As a result of operations on state, it can return ok or partial and set from_next == from and to_next != to. — *end note*]

There is no way to fix C++ streams

- Can remove restrictions…?
  - What about APIs that depended on this upstream?
  - Risky as all get-out (silent, still-compiling change)

- Virtual function hell

- Non-virtual functions have bad interface…!
  - Current landscape status: pretty hosed

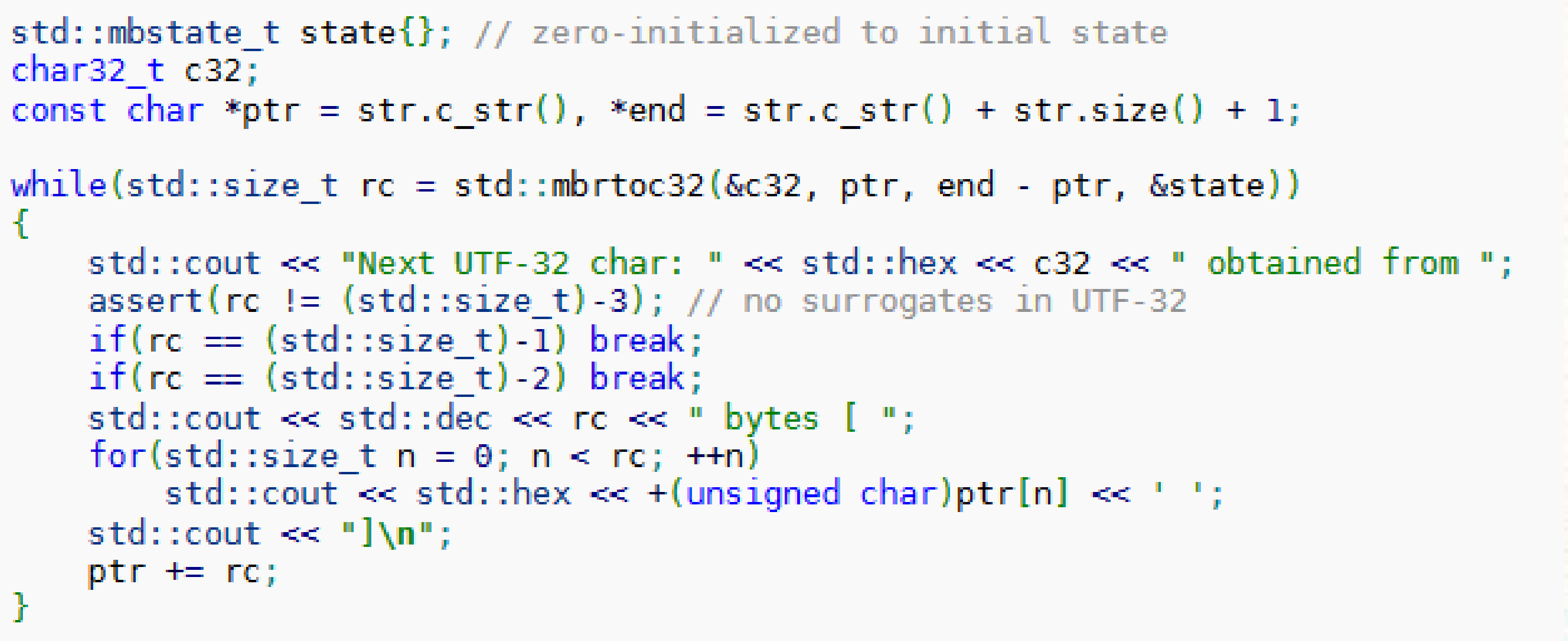
C is no better

- Singular code unit conversion functions
  - `mb(r)towc`, `wc(r)tomb`, `c8/16/32rtomb`, `mbrtoc8/16/32`
  - Explicitly wrong for multibyte encodings (e.g., anything that is not UTF32-alike)
  - "r" means "restartable"; preserves state between calls, potentially helpful for multi code unit encodings (UTF8, UTF16).

- Multi code unit conversion functions
  - Do not exist for c8/16/32…

Well, just loop!

- How hard can it be? It’s C! We’ve got…
  - Pointers!
  - Error codes!
  - No exceptions!
  - Easy 😎



```
std::mbstate_t state{}; // zero-initialized to initial state
char32_t c32;
const char *ptr = str.c_str(), *end = str.c_str() + str.size() + 1;

while(std::size_t rc = std::mbrtoc32(&c32, ptr, end - ptr, &state))
{
    std::cout << "Next UTF-32 char: " << std::hex << c32 << " obtained from ";
    assert(rc != (std::size_t)-3); // no surrogates in UTF-32
    if(rc == (std::size_t)-1) break;
    if(rc == (std::size_t)-2) break;
    std::cout << std::dec << rc << " bytes [ ";
    for(std::size_t n = 0; n < rc; ++n)
        std::cout << std::hex << +(unsigned char)ptr[n] << ' ';
    std::cout << "]\n";
    ptr += rc;
}
```

🐌Slow🐌

- No SIMD possibilities
  - Potentially hard compiler barrier for being in a DLL; no chance to vectorize
  - Restartable versions, even though the conversion might not be stateful

- And the API? Well…

[[digression]]

```
namespace std {

    int mbtowc(
        wchar_t* pwcs,
        const char* mbs,
        size_t num

    );

}
```

[[digression]]

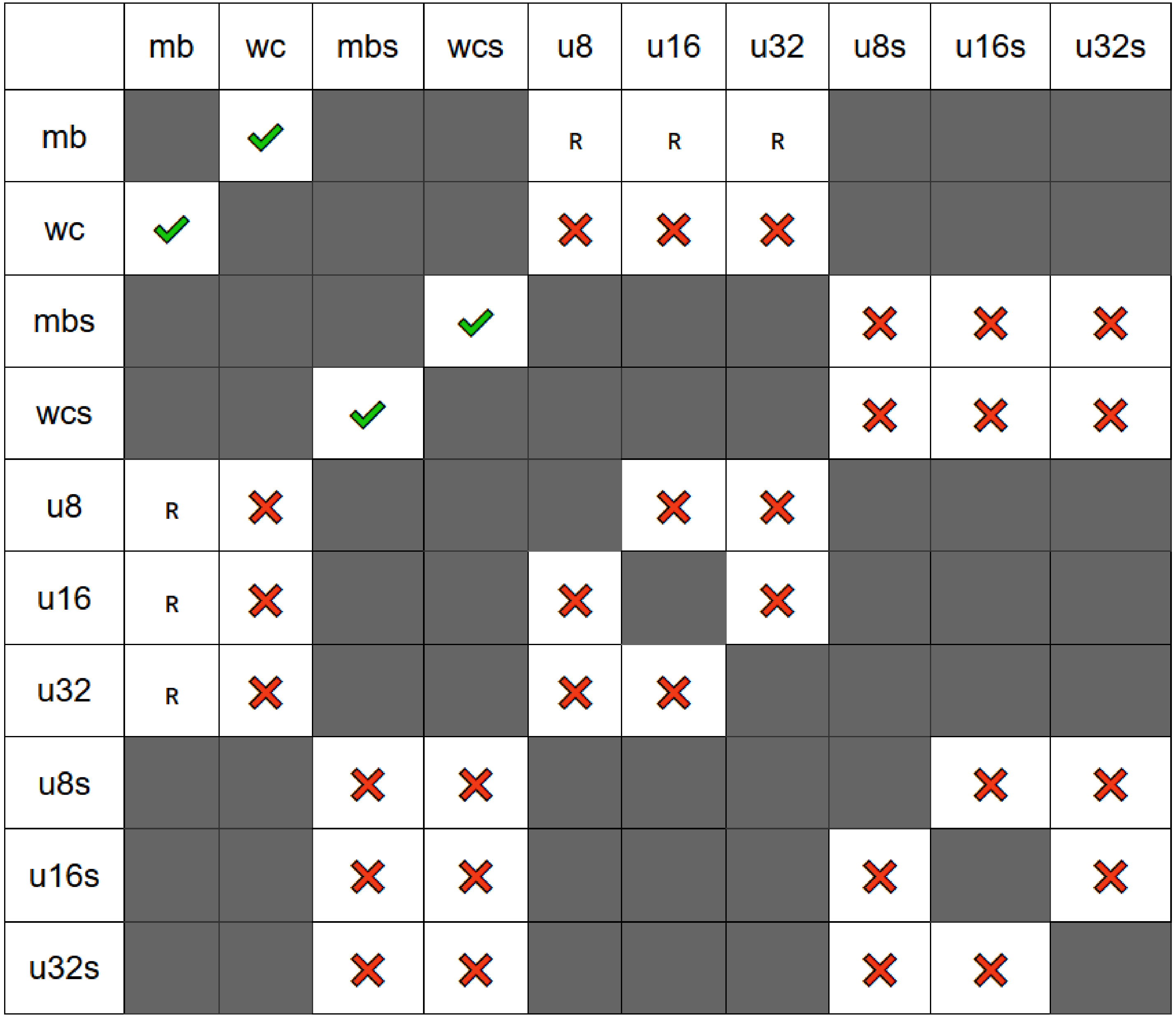
- Stop using pointers to represent optional values in interfaces.
  - `wchar_t*` is supposed to be a wide string
  - But it’s a pointer to a single, optional `wchar_t` instead

- Conventions are great!
  - Until they are wrong.
  - C gets a pass: C++ interfaces don’t.
  - Don’t use pointers for optional values in C++ interfaces.

[[end digression]

Critically Missing APIs

- `{column}(r)to{row}`
  - `mbsrtoc8s`
- Null terminated source strings too
  - Need sized versions

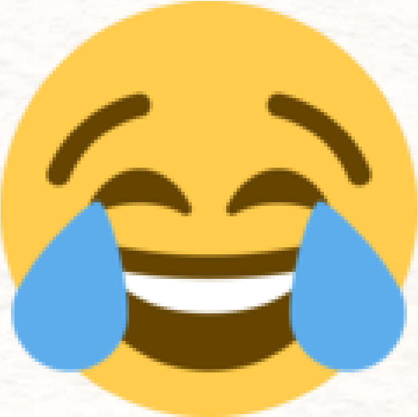


	mb	wc	mbs	wcs	u8	u16	u32	u8s	u16s	u32s
mb	▨	✓	▨	▨	R	R	R	▨	▨	▨
wc	✓	▨	▨	▨	✗	✗	✗	▨	▨	▨
mbs	▨	▨	▨	✓	▨	▨	▨	✗	✗	✗
wcs	▨	▨	✓	▨	▨	▨	▨	✗	✗	✗
u8	R	✗	▨	▨	▨	✗	✗	▨	▨	▨
u16	R	✗	▨	▨	✗	▨	✗	▨	▨	▨
u32	R	✗	▨	▨	✗	✗	▨	▨	▨	▨
u8s	▨	▨	✗	✗	▨	▨	▨	▨	✗	✗
u16s	▨	▨	✗	✗	▨	▨	▨	✗	▨	✗
u32s	▨	▨	✗	✗	▨	▨	▨	✗	✗	▨

Fine. Fine! C and C++ are terrible.

But we have third party libraries!

:joy:



More Seriously…

- ICU has, and likely will always be, sort of a 🗑🔥 in terms of API
  - Still the most fully featured
  - Hand-optimized by many experts

- Several ad-hoc libraries for conversion, but lacking the chops to do the Unicode algorithms
  - Various states of maintenance

Current Libraries

- Boost.Text
  - From prominent boost contributor Zach Laine
  - Handles encoding, decoding, and many higher-level Unicode algorithms
    - Bidi, Segmentation, Collation, …
- libogonek, text_view
  - Libraries from Robot M. Fernandes and Tom Honermann, respectively
  - Handle encoding / decoding (libogonek has a lot more features)

👍 / 👎

- Boost.Text
  - 👍: Covers the widest variety of Unicode algorithms
  - 👍: Uses STL concepts, pretty good speed
  - 👎: Fixed internal representation and normalization forms

- libogonek
  - 👍: Provides many Unicode algorithms, including normalization
  - 👍: Concept-driven design with iterators and decent speed
  - 👎: not recently maintained, `[iterator, iterator)` size blowup

The [iterator, iterator) 💣

-
  ```
  ogonek::detail::grapheme_iterator<
        ogonek::detail::normalize_iterator<
                ogonek::detail::decode_iterator<…>,
        …>,
  …> segmenting_iterator first(…), last(…);
  ```

- Enormous iterator hierarchy: 256+ byte iterators
  - Needed begin + end inside decode iterators;
  - Each iterator was paired with another iterator of the same type;
  - Lots of state management to not overrun / be safe

`std::ranges` to the rescue

- `[decode_iterator, decode_sentinel)`
  - No more book-keeping on both iterators
  - Separate types

`std::ranges` to the rescue

- `[decode_iterator, decode_sentinel)`
  - No more book-keeping on both iterators
  - Separate types

- All information on the iterator half only
  - Sentinel comparison just checks begin/end iterators inside a single `decode_iterator`

Other APIs

- Firefox / Chrome codebases
  - 👍: Extremely fast
  - 👉: Pointer-based API, SIMD optimized
  - 👎: (mostly) codebase-specific
    - Except for Henri Sivonen: wrote encoding_rs

- Some APIs are byte-based, some work with code units
  - A lot of APIs only handle pointers, which exclude exotic storage

Let’s Do Better

The Goal 🥅

```
#include <text>
#include <iostream>

int main () {
        using namespace std::literals;

        std::text::u8text my_text = std::text::transcode(
                "안녕하세요 👏"sv,
                std::text::utf8{}
        );

        // prints 안녕하세요 👏 to a capable console
        std::cout << my_text << std::endl;

        return 0;
}
```

std :: text :: text; a Container Adaptor

```
namespace std::text {

        template <typename Encoding,
                typename NormalizationForm = nfkc,
                typename Container = std::basic_string<…>,
                …
        >
        class basic_text;

}
```

Similarly; std::text::text_view

```
namespace std::text {

     template <typename Encoding,
          typename NormalizationForm = nfkc,
          typename Range = std::basic_string_view<…>,
          …
     >
     class basic_text_view;

}
```

Convenience aliases, as usual

```
namespace std::text {

      using u8text = basic_text<utf8>;
      using u16text = basic_text<utf16>;
      using u32text = basic_text<utf32>;
      using wtext = basic_text<wide_execution>;
      using text = basic_text<utf8>;

      using u8text_view = basic_text_view<utf8>;
      using u16text_view = basic_text_view<utf16>;
      using u32text_view = basic_text_view<utf32>;
      using wtext_view = basic_text_view<wide_execution>;
      using text_view = basic_text_view<utf8>;

}
```

Reality Check

```
namespace std::text {

      using u8text = basic_text<utf8>;
      using u16text = basic_text<utf16>;
      using u32text = basic_text<utf32>;
      using wtext = basic_text<wide_execution>;
      using text = basic_text<narrow_execution>;

      using u8text_view = basic_text_view<utf8>;
      using u16text_view = basic_text_view<utf16>;
      using u32text_view = basic_text_view<utf32>;
      using wtext_view = basic_text_view<wide_execution>;
      using text_view = basic_text_view<narrow_execution>;

}
```

Container/View Adaptors?

- Flexible
  - Expects a *SequenceContainer*
  - Works with std::deque<T>, llvm::SmallVector<T>, __gnu_cxx::rope<T>, gap_buffer, …

- Cannot afford to reinvent everyone's API with a new string type!
  - ICU does it with `UnicodeString`
  - Not at all fun

Nothing Changes

```cpp
// thing.hpp

inline std::string do_the_thing(std::string totes_utf8);

// thing.cpp

std::string do_the_thing(std::string totes_utf8) {
     // the stuff, manually
}
```

Nothing Changes (for the user)

```cpp
// thing.hpp

inline std::string do_the_thing(std::string totes_utf8);

// thing.cpp

using my_u8text = std::text::basic_text<basic_utf8<char>>;

std::string do_the_thing(std::string totes_utf8) {
    my_u8text deffo_utf8(std::move(totes_utf8));
    // the stuff, better :D
    return std::move(deffo_utf8.base());
}
```

The Goal – Networking 🥅

```
namespace beast = boost::beast;         // from <boost/beast.hpp>
namespace http = beast::http;           // from <boost/beast/http.hpp>
using tcp = boost::asio::ip::tcp;       // from <boost/asio/ip/tcp.hpp>
using results_type = tcp::resolver::results_type;

class session : public std::enable_shared_from_this<session> {
        /* … */
        http::request<http::empty_body> req_;
        std::vector<std::byte> res_body_;
        http::response<http::vector_body<std::byte>> res_;
        std::u8string converted_body_;

        /* … */
        void on_connect(beast::error_code ec, results_type::endpoint_type);
        void on_resolve(beast::error_code ec, results_type results);
        void on_read(beast::error_code ec, std::size_t bytes_transferred);
};
```

The Goal – Networking 🥅

```
void session::on_read(beast::error_code ec, std::size_t bytes_transferred) {
        if (ec) {
                log_fail(ec, u8"read failed");
                return;
        }

        std::span<std::byte> bytes(res_body_.data(), bytes_transferred);

        /* continued … */
}
```

The Goal – Networking 🥅

```cpp
void session::on_read(beast::error_code ec, std::size_t bytes_transferred) {
	/* … from previous slide */
	std::ranges::unbounded_view output(
		std::back_inserter(converted_body_)
	);

	std::text::encoding_scheme<std::text::utf16,
		std::endian::big> from_encoding{};
	std::text::utf8 to_encoding{};

	// transcode from bytes, into unbounded output
	std::text::transcode(bytes, output, from_encoding, to_encoding);

	std::clog << converted_body_ << std::endl;
}
```

Keep this dream in mind…

Digging Deep

Or: “How do we get from here to there?”

Foundation: Encoding objects

- Encoding is a concept that a class type can satisfy
  - It has some required (and optional) operations

- Serves as the foundational building block
  - Encodes and decodes one code point at a time
  - Member types and static member variables to dictate some useful defaults

Helper Types: Basics

```
struct empty_struct {};

using u8_span = std::span<char8_t>;
using u16_span = std::span<char16_t>;
using u32_span = std::span<char32_t>;

enum class encoding_errc : int {
     ok = 0x00,
     invalid_sequence = 0x01,
     insufficient_output_space = 0x02,
};
```

Helper Types: Result Types

```
struct decode_result {
     u8_span input;
     u32_span output;
     empty_struct& state;
     encoding_errc error_code;
};


struct encode_result {
     u32_span input;
     u8_span output;
     empty_struct& state;
     encoding_errc error_code;
};
```

Helper Types: Error Handlers

```
using decode_error_handler = std::function_ref<
     decode_result(utf8&, decode_result)
>;

using encode_error_handler = std::function_ref<
     encode_result(utf8&, encode_result)
>;
```

An example encoding object

```
struct utf8 {
      using code_unit                = char8_t;
      using code_point               = char32_t;
      using state                    = empty_struct;
      using is_decode_injective = std::true_type;
      using is_encode_injective = std::true_type;
      static constexpr inline std::size_t max_code_points = 1;
      static constexpr inline std::size_t max_code_units = 4;

      encode_result encode(u8_span input, u32_span output,
            state& current, encode_error_handler error_handler);

      decode_result decode(u32_span input, u8_span output,
            state& current, decode_error_handler error_handler);
};
```

(Gory details for the standard)

```
template <typename _CharT = char8_t>
struct basic_utf8 {
        using code_unit           = _CharT;
        using code_point          = char32_t;
        using state               = empty_struct;
        using is_decode_injective = std::true_type;
        using is_encode_injective = std::true_type;
        static constexpr inline std::size_t max_code_points = 1;
        static constexpr inline std::size_t max_code_units = 4;

        template <typename __InputRange, typename __OutputRange,
                typename __ErrorHandler>
        static constexpr auto encode(__InputRange&& __input, __OutputRange&& __output,
                state& __s, __ErrorHandler&& __error_handler);

        template <typename __InputRange, typename __OutputRange,
                typename __ErrorHandler>
        static constexpr auto decode(__InputRange&& __input, __OutputRange&& __output,
                state& __s, __ErrorHandler&& __error_handler);
};
```

An example encoding object

```
struct utf8 {
      using code_unit            = char8_t;
      using code_point           = char32_t;
      using state                = empty_struct;
      using is_decode_injective = std::true_type;
      using is_encode_injective = std::true_type;
      static constexpr inline std::size_t max_code_points = 1;
      static constexpr inline std::size_t max_code_units = 4;

      encode_result encode(u8_span input, u32_span output,
            state& current, encode_error_handler error_handler);

      decode_result decode(u32_span input, u8_span output,
            state& current, decode_error_handler error_handler);
};
```

`code_unit`

- The units that can be composed to create one meaningful point of information
  - Can take 1 to N `code_unit`s to make a single meaningful point of information
  - UTF8: 1-4 code units, 1-4 bytes
  - UTF16: 1-2 code units, 2-4 bytes
  - UTF32: 1 code unit, 4 bytes
  - ASCII: 1 code unit, 1 byte

`code_point`

- An indivisible unit representing an unambiguous bit of information
  - Can represent every single indivisible unit of information in the character repertoire
  - Gb18030, UTF8, UTF16, UTF32: produce 1 code point
  - ASCII: produces 1 code point (!!)

- Generally, represented as `char32_t`

state

- State is used for stateful encodings
  - E.g., ISO-2022 JP
  - Can encode “shift states” or “modes” that change how later characters are processed
- Trifecta of Unicode Encodings are not stateful
  - So they are just an empty struct

encode, decode

- 2 functions which take in an input range, an output range, the current state, and an error handler

- Computes exactly one – and only one – indivisible unit of information:
  - multiple code units → one code point;
  - one code point → multiple code units;
  - modify state + one of the above;
  - or, output error

Uh… “output range”?

- A better abstraction for handling sized output: safe!

```
std::u8string input = u8“MΛΓ ΓλЄS ÏTΛN, NI MIS ΥΠ NЗΛN BRIΓΓIΨ.”;
std::u32string output(16, U‘\0’);
std::text::utf8 encoding{};
std::text::utf8::state state{};

auto result = encoding.decode(
     input, std::span(output),
     state, std::text::default_handler{}
); // does not overrun buffer
```

Need speed?

- Ask for it with an unbounded view

```
std::u8string input = u8“𐌼𐌰𐌲 𐌲𐌻𐌴𐍃 𐌹̈𐍄𐌰𐌽, 𐌽𐌹 𐌼𐌹𐍃 𐍅𐌿 𐌽𐌾𐌰𐌽 𐌱𐍂𐌹𐌲𐌲𐌹𐌸.”;
std::u32string output(16, U‘\0’);
std::text::utf8 encoding{};
std::text::utf8::state state{};

auto result = encoding.decode(
     input, std::ranges::unbounded_view(output.data()),
     state, std::text::default_handler{}
); // may overrun buffer
```

[[dangerous]] Need more speed?

- Use the “I’m smarter than you”, UB error handler

```
std::u8string input = u8“ϺɅΓ ΓλƐS ῙTɅN, NI MIS ϒՈ NϽɅN BꞦIΓΓIψ.”;
std::u32string output(16, U‘\0’);
std::text::utf8 encoding{};
std::text::utf8::state state{};

auto result = encoding.decode(
     input, std::ranges::unbounded_view(output.data()),
     state, std::text::assume_valid_handler{}
); // may overrun buffer and explode code
```

`is_(decode|encode)_injective`

- Communicates whether transformation is lossy
  - marks a `decode/encode(…)` operation as being fundamentally lossy
  - Injective: one-to-one mapping that preserves distinctness

- Solves ASCII problem:
  - ascii has only 1 code unit, 1 code point
  - cannot represent more than 7 bits of information
  - `U"🐶"` – cannot be represented!

`is_(decode|encode)_injective`

- Not used directly in low-level encoding object interface
  - Used to require error handler for any lossy conversions

- `static_assert` failure when:
  - encoding/decoding not injective;
  - code point types of encodings are convertible to one another;
  - or, encoding/decoding is injective one way, but not another.

Injective properties: safety control

- Narrowing is generally evil and causes (subtle) bugs
  - std::size_t $\boxed{\rightarrow}$ int; uint32_t $\boxed{\rightarrow}$ int16_t

```
struct ascii {
     // ASCII → Unicode, fine
     using is_decoding_injective = std::true_type;
     // Unicode → ASCII, not fine
     using is_encoding_injective = std::false_type;
     using code_unit = char;
     /* … */
};
```

Bad conversion

```
std::string ascii_emoji = std::text::encode(
     U“🐶”,
     std::text::ascii{}
);
```

Bad conversion

```
std::string ascii_emoji = std::text::encode(
    U“🐶”,
    std::text::ascii{}
); // Compiler Error: this is a lossy conversion
```

Want to Narrow? Be *Explicit*

```
std::string ascii_emoji2 = std::text::encode(
     U“🐶”,
     std::text::ascii{},
     std::text::default_handler{}
); // Compiler Error

std::string ascii_emoji3 = std::text::encode(
     U“🐶”,
     std::text::ascii{},
     std::text::replacement_handler{}
); // … Well, I mean, you asked for it!
```

Standard Encodings

```cpp
template <typename Char>
class basic_utf8;
template <typename Char>
class basic_utf16;
template <typename Char>
class basic_utf32;

class ascii;
class narrow_execution;
class wide_execution;
using utf8 = basic_utf8<char8_t>;
using utf16 = basic_utf16<char16_t>;
using utf32 = basic_utf32<char32_t>;
```

Beyond the Objects

What simple encoding objects enable?

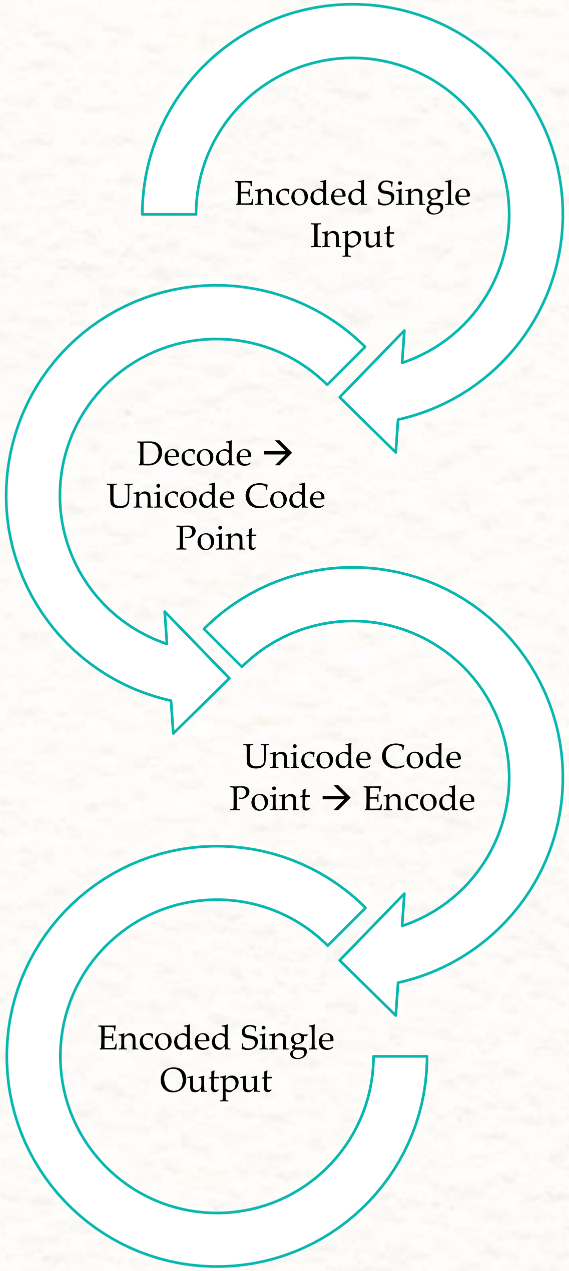
Need for Speed

- Performance is correctness.
  - No way around it.

- 3-speed approach
  - Slowest Path (round-trip transcoding, lazy)
  - Faster Path (direct transcoding, no round-tripping, lazy)
  - Fastest Path (bulk processing, eager consumption)

Slowest

- Works for Everything™
- Ideal for disparate encodings
  - SHIFT-JIS to GB18030 to EBCDIC
- Roundtrips through Unicode
- Converts one codepoint at a time



Slowest: Explicit Encode, Decode

- Roundtrip from encoding to decoding through common code point
  - One-by-one encoding and checking
  - Safe, scalable

- Uses range types, like `std::text::text_view`
  - Take advantage of `std::ranges` allowing begin/end to be different
  - Reduces problem of layering iterators

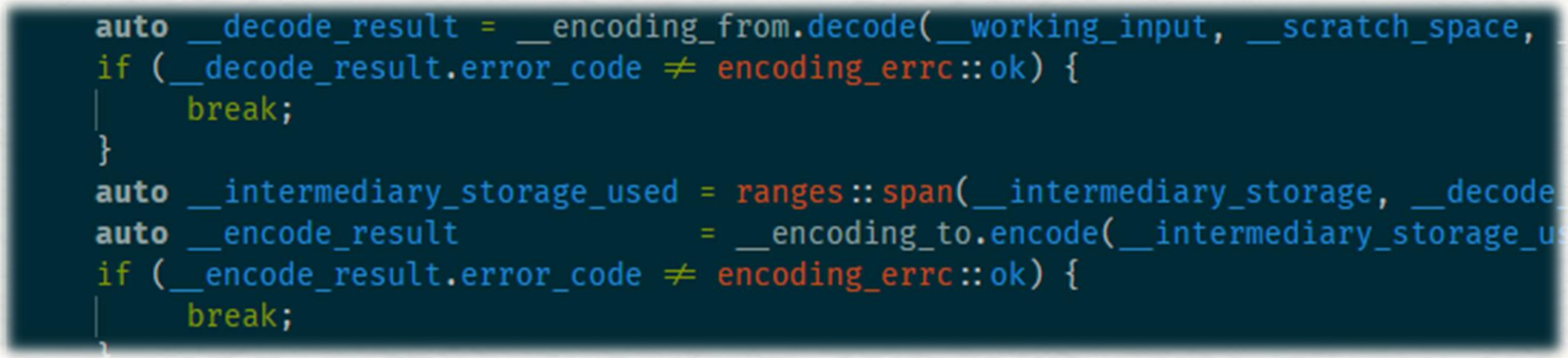
Slowest: lazy `transcode_view` 🐢

- `(encode|decode|transcode)_view` wraps a range
  - takes encoding objects as template parameters (From, To)
  - Always round-trip converts

```
std::text::transcode_view<std::string_view,
      std::text::narrow_execution,
      std::text::utf32
> lazy = “こんばんわ。”;

for (char32_t my_utf32_cp : lazy_view) {
      /* do as you wish */
}
```

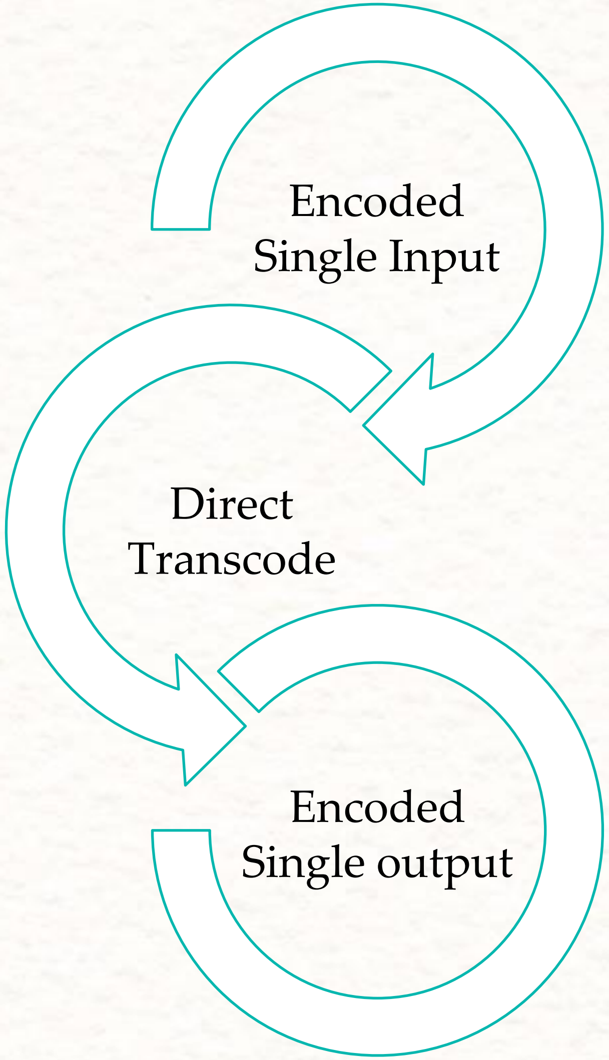
Slowest: a code Snapshot 🐢



```
auto __decode_result = __encoding_from.decode(__working_input, __scratch_space,
if (__decode_result.error_code ≠ encoding_errc::ok) {
    break;
}
auto __intermediary_storage_used = ranges::span(__intermediary_storage, __decode
auto __encode_result             = __encoding_to.encode(__intermediary_storage_u
if (__encode_result.error_code ≠ encoding_errc::ok) {
    break;
}
```

Faster Path

- Specific to a direction and a pair of encodings
- Ideal for encodings where we know there are fast conversions (UTF8 🔄 UTF32, UTF16 🔄 UTF32, etc.)
- Converts directly without necessarily going to an intermediate
- One at a time conversion



Encoded Single Input

Direct Transcode

Encoded Single output

Faster: `text_transcode_one` 🐇🏃

- ADL function a user writes to hook transformation
  - Allows users to override standard roundtrip behavior

- Faster, but not fastest:
  - Still a one-by-one transformation
  - Still expects one individual unit of information to be consumed

Faster: text_transcode_one 🐇🏃

```
void text_transcode_one(
      u8_span input, u32_span output,
      std::text::utf8& input_enc, std::text::utf32& output_enc,
      empty_struct& input_state, empty_struct& output_state,
      decode_error_handler decode_handler,
      encode_error_handler encode_handler
) {
      // just… decode!
      (void)output_encoding;
      (void)encode_handler;
      (void)output_state;
      input_enc.decode(input, output, input_state, decode_handler);
}
```

Faster: `text_transcode_one` 🐇🏃

- Write one for UTF16 🔄 UTF8, or GB 🔄 GB18030

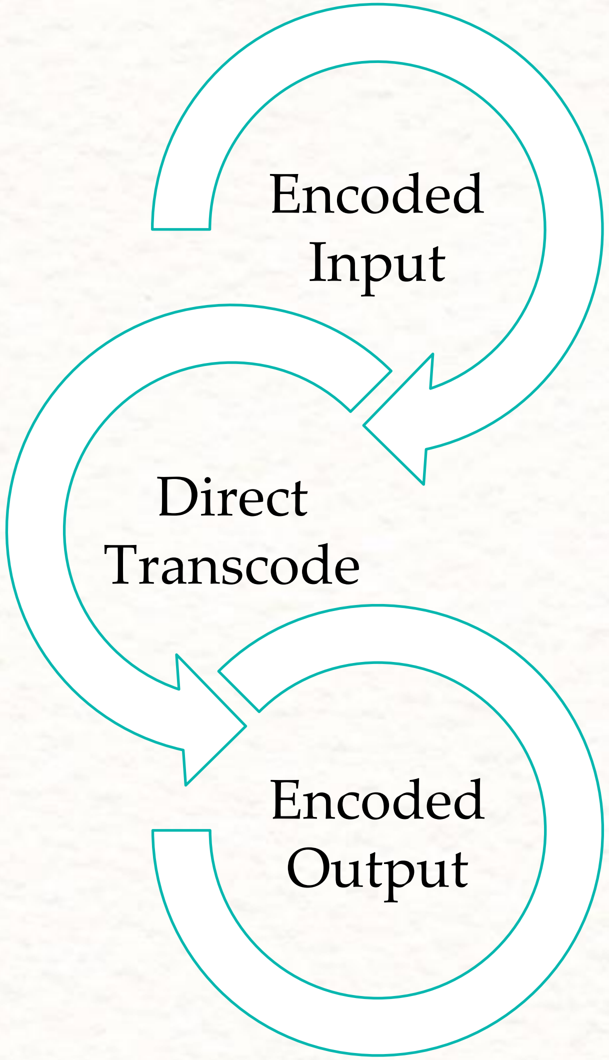
- Important: code still looks the same from before!

```
std::text::transcode_view<
      std::u8string_view, std::text::utf8, std::text::utf32
> lazy_view = u8“🐶 woof!”;

for (char32_t my_utf32_cp : lazy_view) {
      /* do as you wish! */
}
```

Fastest Path

- Specific to a direction and a pair of encodings
- Bulk transcode converts directly without necessarily going to a code point
- Bulk conversion (SIMD, etc.)
- Typically done on *ContiguousRange*s



Encoded Input

Direct Transcode

Encoded Output

Fastest: eager function 🏎

- Absolute fastest path
  - Free functions: encode, decode, transcode
  - convert ***as much as possible*** – not restricted to “one code point at a time”

```
std::text::u8text my_string = std::text::transcode(
        u“𐌼𐌰𐌲 𐌲𐌻𐌴𐍃 𐌹̈𐍄𐌰𐌽, 𐌽𐌹 𐌼𐌹𐍃 𐍅𐌿 𐌽𐌳𐌰𐌽 𐌱𐍂𐌹𐌲𐌲𐌹𐌸.”,
        std::text::utf8{}
);
```

Fastest: Customization

- People outside the standard will have encodings and encoding conversion pairs they care about
  - Takes pressure of implementers to have to provide amazing QoI

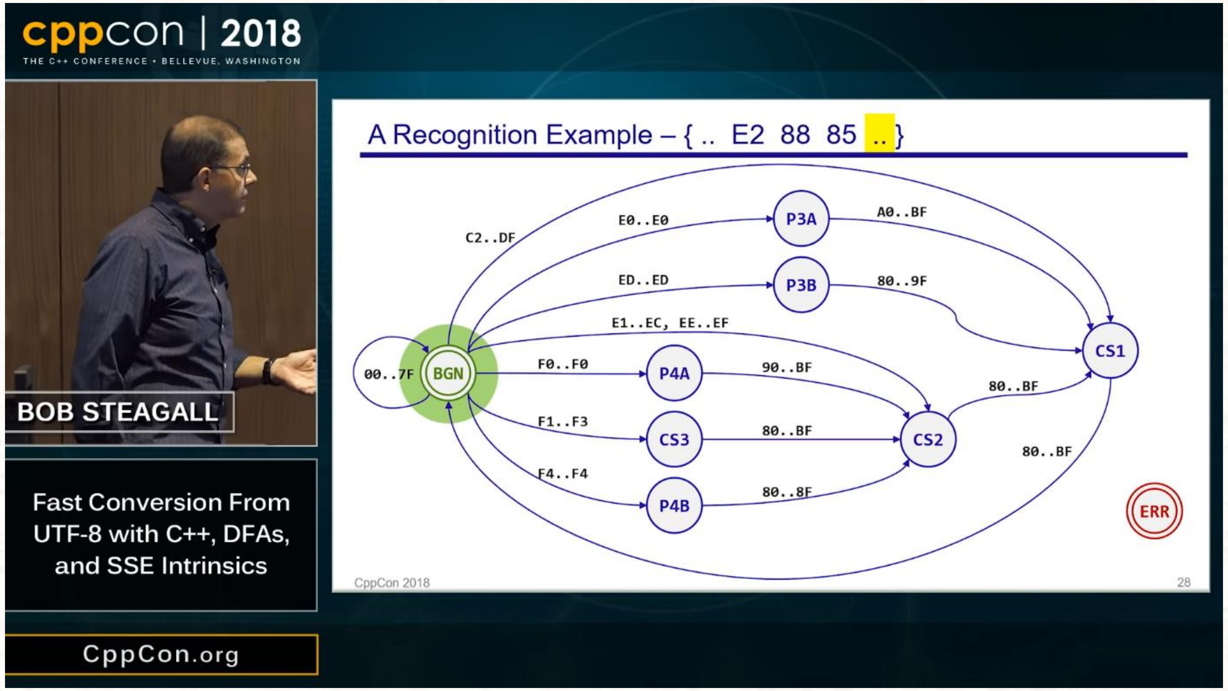
- Substitute in proprietary conversions for the 90% cases already optimized for
  - The rest will still work through the “Faster” and “Slower” paths

Fastest: text_[en/de/trans]code

```
void text_transcode(
      u8_span input, u16_span output,
      std::text::utf8, std::text::utf16,
      empty_struct&, empty_struct&,
      decode_error_handler decode_handler,
      encode_error_handler
) {
      std::ptrdiff_t result = UtfUtils::SseConvert(
            input.begin(), input.end(), output.begin());
      if (result == -1) {
            decode_handler(…);
      }
}
```

Fastest Path: In your Hands

- https://github.com/BobSteagall/utf_utils https://www.youtube.com/watch?v=5FQ87-Ecb-A



cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

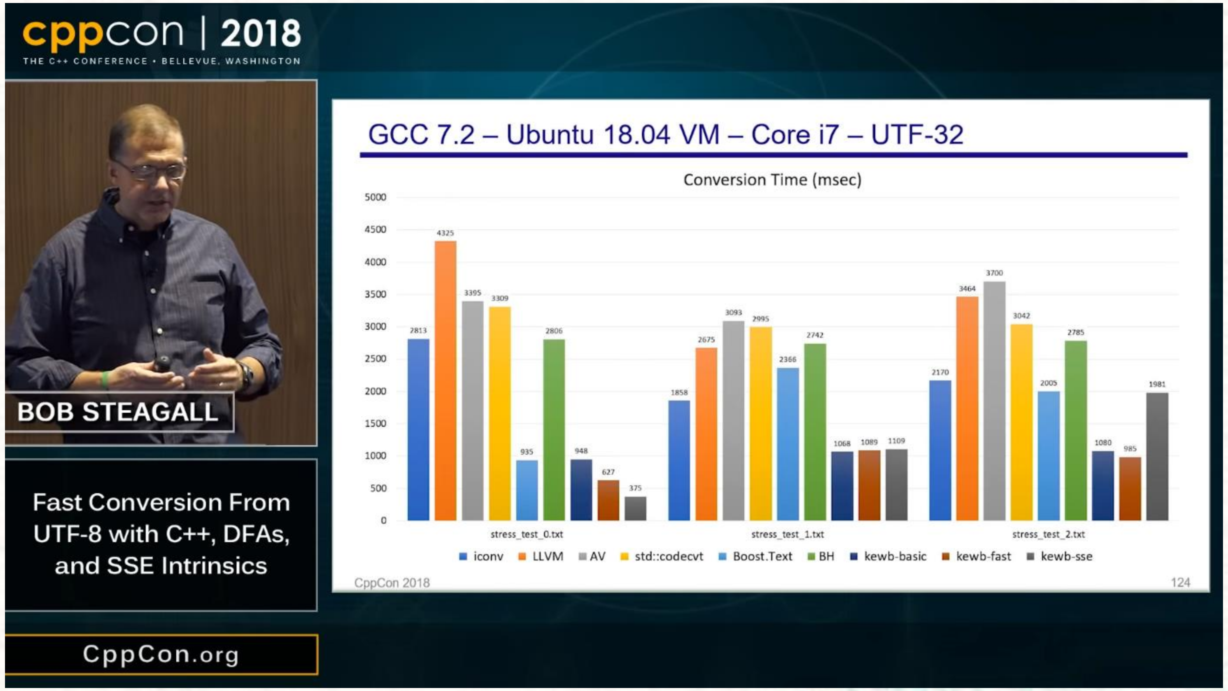
BOB STEAGALL

Fast Conversion From UTF-8 with C++, DFAs, and SSE Intrinsics

CppCon.org

A Recognition Example – { .. E2 88 85 .. }

CppCon 2018 28



cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

BOB STEAGALL

Fast Conversion From UTF-8 with C++, DFAs, and SSE Intrinsics

CppCon.org

GCC 7.2 – Ubuntu 18.04 VM – Core i7 – UTF-32

CppCon 2018 124

Accelerating Development

> This foundational work is needed in C++23 to enable non-experts to write ‘hello 🌎’.

— Tom Honermann
Chair, Study Group 16

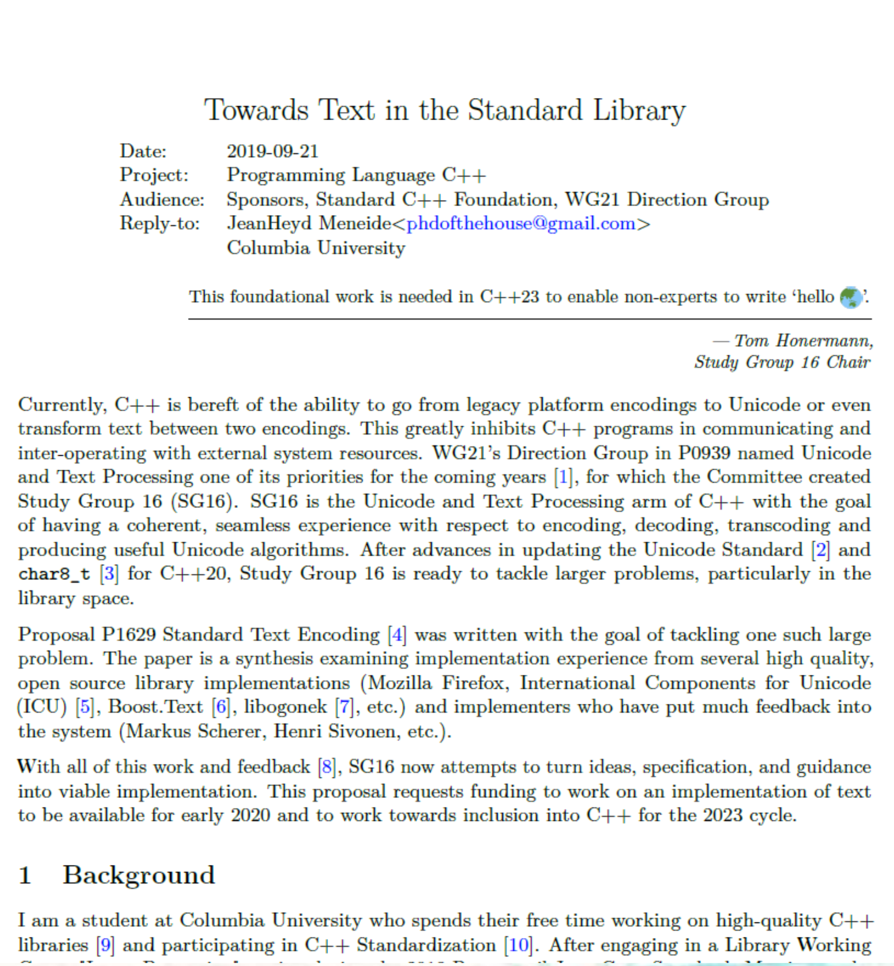
P1629 – Standard Text Encoding

- A way to get the typically digitally underserved, served
  - Making it simple to perform encoding conversions
- P1629 needs an implementation
  - A lot of work
  - Current Feel of the Committee: “Library implementation or get out.”

Sponsorship Necessary

- Everyone complains about Text and Unicode support
  - Even if you cannot do the work, you can help fund it.

- Goal: extended implementation by 2020
  - Early in the C++23 cycle



Towards Text in the Standard Library

Date: 2019-09-21
Project: Programming Language C++
Audience: Sponsors, Standard C++ Foundation, WG21 Direction Group
Reply-to: JeanHeyd Meneide<phdofthehouse@gmail.com>
Columbia University

> This foundational work is needed in C++23 to enable non-experts to write 'hello 🌏'.
>
> — *Tom Honermann, Study Group 16 Chair*

Currently, C++ is bereft of the ability to go from legacy platform encodings to Unicode or even transform text between two encodings. This greatly inhibits C++ programs in communicating and inter-operating with external system resources. WG21's Direction Group in P0939 named Unicode and Text Processing one of its priorities for the coming years [1], for which the Committee created Study Group 16 (SG16). SG16 is the Unicode and Text Processing arm of C++ with the goal of having a coherent, seamless experience with respect to encoding, decoding, transcoding and producing useful Unicode algorithms. After advances in updating the Unicode Standard [2] and `char8_t` [3] for C++20, Study Group 16 is ready to tackle larger problems, particularly in the library space.

Proposal P1629 Standard Text Encoding [4] was written with the goal of tackling one such large problem. The paper is a synthesis examining implementation experience from several high quality, open source library implementations (Mozilla Firefox, International Components for Unicode (ICU) [5], Boost.Text [6], libogonek [7], etc.) and implementers who have put much feedback into the system (Markus Scherer, Henri Sivonen, etc.).

With all of this work and feedback [8], SG16 now attempts to turn ideas, specification, and guidance into viable implementation. This proposal requests funding to work on an implementation of text to be available for early 2020 and to work towards inclusion into C++ for the 2023 cycle.

1 Background

I am a student at Columbia University who spends their free time working on high-quality C++ libraries [9] and participating in C++ Standardization [10]. After engaging in a Library Working

S G 16

Join us online, at the mailing list, in our repositories, open teleconferences, and more:

https://github.com/sg16-unicode/sg16

Support my work:
https://github.com/users/ThePhD/sponsorship
https://thephd.github.io/support/