Considerations for Unicode algorithms

Document #:	D2773R0
Date:	2023-02-16
Programming Language C++	
Audience:	SG-16
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

As work is ongoing to propose Unicode algorithms for C++ 26, this paper attempts to summarize my thoughts and my implementation experience with some of these algorithms. I aim to provide some background on what these algorithms are and how they operate, as well as give feedback on Zach's Laine excellent work on P2729R0 [7].

TL;DR

This is an opinion piece derived from my own experiences with the implementation and use of Unicode algorithms, and I make the case for the following recommendations

- Most Unicode algorithms should be exposed as views (satisfying the requirements of `<ranges>`)
- Unicode algorithms should compose well with the wide array of views and algorithms in the standard
- Normalization, Clusterization and Casing should be the focus of the initial work (assuming UTF transcoding is standardized; we can't do anything before that)
- Unicode algorithms should operate on code points
- Tailored and non-tailored algorithms should be exposed through similar but separate interfaces as they has different requirements and implementation challenges
- We should have a good idea of locale representation before working on tailored algorithms
- Range adaptor objects (pipe syntax) are the most suitable place to introduce implicit UTF decoding/encoding steps. Having these implicit steps is necessary for usability.
- `char32_t` is the type suitable to represent a code point
- consumption of `char`, `std::bytes` and others should be possible but explicit.
- code unit sequences should be validated by default.

- We should not constraint the interface of non-tailored algorithms to make them implementable using ICU
- ICU4x is most certainly the best answer to tailored algorithms in the long term
- We should avoid exposing assumptions that may change in future Unicode versions
- Optimizing property lookups and cleverly reserving memory for non-sized ranges is how we can provide performance more than competitive with existing frameworks.
- Unicode algorithms do not benefit from in-place transformation and theorical text-specific containers.
- Trying to avoid UTF decoding/encoding does not lead to obvious performance benefits.

Unicode algorithms

Unicode algorithms are algorithms defined by The Unicode Standard and operate on sequences of code points. They fall into the following categories:

- Canonicalization: As abstract characters can have multiple representations in Unicode (mostly for compatibility reasons), Unicode provides "normalization algorithms" that allow two sequences of code points representing the same abstract characters to compare equal. These algorithms are $N- > M$ transformations, and their primary effect is to decompose and recompose (depending on the form) diacritics, Hangul syllables, and other combined characters.
- Clusterization: Algorithms to separate graphemes, words, and sentences. These algorithms are "`group by`"/"`chunk by`" algorithms: they produce a range of ranges given some boundary condition.
- Casing: Unicode provides algorithms to compare sequences of code points ignoring their case, and to transform the case of those sequences.: Upper casing, lower casing, and title casing. Title casing (in which uppercase the first letter of each word) does depend on clusterization to find word boundaries. Like normalization these are $N- > M$ transformations: The process of casing can add or remove code points.
- Collation: The process of sorting Unicode strings.
- Bidirectional text handling: This algorithm determines the directionality of text for the purpose of text rendering. There are other uses for these algorithms, for example as part of the ongoing effort to prevent bidirectionality-based source code spoofing attacks. But on average, this is specific to text rendering.

That's more or less it. In particular, transcoding and other algorithms fall outside of the definition of Unicode algorithms. And Unicode algorithms are only ever applicable to Unicode text.

The full list as it appears in the Unicode standard:

```
Canonical Ordering
Canonical Composition
Normalization
Hangul Syllable Composition
Hangul Syllable Decomposition

Default Case Conversion
Default Case Detection
Default Caseless Matching

Bidirectional Algorithm

Line Breaking Algorithm
Character Segmentation
Word Segmentation
Sentence Segmentation
Hangul Syllable Boundary

Standard Compression Scheme for Unicode

Unicode Collation Algorithm

Hangul Syllable Name Generation
```

For example, normalization cannot be applied to some sequence of code units of a non-Unicode encoding.

This is important as there is a lot of commonality in these algorithms: they have similar behavior, performance profiles, preconditions, implementation challenges, etc. They all operate on Unicode code points.

Tailoring

With the exception of Bidirectional handling and normalization, which are never tailored, all other algorithms can be tailored: that is to say, they can be affected by a locale. For example, `ch` is a single grapheme in a Slovak locale. Similarly, casing behaves differently in different locales, the classical example being i -> İ in Turkish, but 5 or 6 languages are also affected, such as Greek, Dutch, Armenian, German (and the rules for German's ß have changed a few times in recent years).

Unicode provides a specification for the default (non-tailored) version of the algorithms and recommends an implementation to tailor them according to local, tailoring information being provided by the Common Locale Data Repository.

Whether to use a tailored algorithm and which locale to use then is highly use-case dependent.

Ideally, for user-facing uses cases, such as some UI, the data should be rendered per a locale appropriate for the user (which may or may not be the system's locale), and for some other use cases, like searching some data in a database, locale awareness may not be desirable.

Order of operations: Walking before we can run

We have established that there are about five families of algorithms and related utilities that are part of the Unicode standard, and many of these algorithms benefit from locale information. This is an unrealistic amount of work for C++26. In particular, tailored algorithms do depend on CLDR data and are best implemented with a 3rd party library such as ICU/ICU4x. Whereas non-tailored algorithms only depend on code point properties, so have different implementation complexity. But also, the error conditions and the performance profiles are not the same, nor are the requirements we can put on these algorithms. In fact, I'm pretty convinced that we ought to treat the tailored and non-tailored algorithms entirely differently, probably using different types, such that we are able to provide an efficient non-tailored implementation that is usable on all platforms, and later add tailored versions on systems were a CLDR wrapper is usable.

The elephant in the room, of course, is that locale-dependant algorithms do require a locale object, which we don't have. P2020R0 [3] goes into more detail as to why `std::locale` is not the answer here. A strong requirement is that the locale identifier be portable and able to allow the extraction of language and region tags compatible with Unicode, BCP47, the CLDR, and CLDR implementations are such that ICU/ICU4x, which `std::locale` is unsuitable for. Moreover, P2020R0 [3] establishes that locale facilities should not be implicit or defaulted as there more often than not a mismatch between the system locale and the intent of the application/function.

As such, I recommend that we should focus on non-tailored algorithms in the 26 cycle, while a suitable locale object materializes. I'm fairly convinced that the long-term solution is to rely on ICU4x for locale-dependent facilities.

It could be tempting to observe that for a given Unicode algorithm, such as casing, there is a limited number of usual tailoring. We could design an interface that takes, not a locale identifier object, but some kind of enum that would list these different known tailoring. I would be rather concerned with such an approach: It is a short-term solution that adds a large amount of implementation complexity as implementers would have to understand, implement and maintain the specificities of various locales, and maintain them. It would also put an additional burden on the committee to maintain this list of behaviors, and worse specify them. Locale-dependant behavior need not be stable as the CLDR improves, or government recommendations and usage and customs change.

Worse, we run the risk of having second-class locales. Sure, SG-16 does love to talk about the dotted İ, but are we really crossing our İs if we support Turkish correctly, but fail to equally support Lithuanian or Tibetan?

Users would have to map std::local to these enum themselves, which is fraught with peril as no one specify the format of ""standard"" locale identifiers.

I should be very clear that I am not saying tailoring is not useful, it certainly is and we should work toward a solution that is workable, one that meets user expectations and has a reasonable implementability.

We are not there yet, so let's focus on untailored algorithms first, and go from there.

Similarly, we ought to prioritize which algorithms we work on.

Collation in addition of being hugely complex, only really makes sense when tailored, so we should punt that problem to later.

BiDi is not universally useful - applications that do need it are usually based on rendering frameworks such as Qt, win32, cocoa, or web browsers that already have the necessary support. We could make the case for standardization given the applicability to source code spoofing, for example, however, the implementation complexity of this algorithm should make it a lesser priority.

On the other hand, normalization is very fundamental to any Unicode handling and should be the focus of an initial effort - despite a moderate complexity.

Casing (both transformations and comparison), are common operation in both user-facing and server applications dealing with for example user databases, and seems a good candidate for standardization, especially as it would allow us to deprecate some of the `<ctypes>` facilities, with prejudice (their interface cannot be used correctly).

Clusterization presents little complexity, and we should establish grapheme as a vocabulary type. Plus, word clusterization is a prerequisite to Title Casing, and grapheme clusterization is already implemented even if the implementation is hidden behind `std::format`.

Sentence clusterization may have fewer uses outside of text editors and text layout engines, so we should not prioritize it, but at the same time, it shares a lot of similarities and implementation with word clusterization so it's a rather cheap facility to add, once we have everything else.

Understanding Unicode algorithms

At a high level, Unicode algorithms act on Unicode code points. They look at various properties of the Unicode point at the current position (each Unicode algorithm relies on one or more - often dedicated - properties), or of its neighbors (either the code points before, or the code points after), and, for algorithms that are transformations produce a new code point, or set of code points.

This has interesting consequences:

- All of these algorithms have similar interfaces, in particular, they all operate on code points.
- They can operate on forward iterators and bidirectional iterators. Because they require lookaheads, and lookbacks, they can usually not operate on non-forward iterators,

and because they are $N- > M$ transformations, they cannot produce random access iterators.

- They can all provide bidirectional access which - at the cost of some implementation complexity - allows a greater composability with existing algorithms.

Failure modes and preconditions

Unicode algorithms have a fairly wide contract. Their one precondition is that they are defined on code points, i.e. integers in the range 0-10FFFF. In particular, they are well defined for lone surrogate, reserved, unassigned and PUA code points. Note that producing lone surrogates is a violation of transcoding post conditions, and transcoding facilities should be aware of that.

Normalization could allocate, which could introduce additional failure modes (more in that later), and, depending on implementation, tailored algorithms could potentially read data stored out of process or use third party libraries with their own, sometimes opaque, failure modes.

On text containers

The $N- > M$ nature of Unicode transformations, in particular, is important. In-place Unicode transformations cannot be implemented efficiently: because the input and output size, in the arbitrary case, do not match, a hypothetical container that would provide in-place Unicode transformations would be forced to do insertions and therefore allocations in the middle which is not efficient. This, among other considerations, negates previous discussions in SG16 about adding some ”text” containers.

It doesn’t mean that there isn’t value in some containers that would be targeted toward text manipulation. For example, ropes are useful based on the observation that text editors perform a lot of insertions and deletions in the middle. But this has more to do with the nature of text editors than it has to do with any Unicode algorithms. And I’m not sure how much we would want to put text-editor-specific building blocks in the standard libraries.

Unicode algorithms are transformations. There is usually very little value in trying to maintain an invariant such as ”uppercased” (in which locale?), as it would be, in the general case, less efficient than building whatever text, and then uppercase it as a whole (again because in the general case, such transformation is $N- > M$).

Moreover, given the whole set of Unicode algorithms, plus related facilities (such as decoding), which invariants should such a container maintain? And, if the answer is none, why would we want more containers? (Standardizing and maintaining containers is not a small endeavor for the committee, implementers, and our users).

So it is clear to me that Unicode algorithms should be implemented as such: as algorithms. Or should they?

Unicode algorithms: views first

There are algorithm-specific considerations but in the general case, we have 3 choices:

- We can provide old-style algorithms
- We can provide views satisfying the ranges requirements
- We can entertain doing both.

In my experience, providing views for transformations is enough, easier, and consistent with the direction C++ has been persuing over the past 2 cycles, so it should hopefully be more intuitive for users.

Observations applicable to all Unicode algorithms

`char32_t` as code point type

Unicode algorithms operate, without exception on code points. I've seen some proposals (such as P0244R2 [1]) in the past trying to define either a type or a concept for a Unicode code point. I think this is a mistake. A concept would imply that there would be several ways to represent a code point, which would hurt composability and add complexity. A type would imply that we do not already have the necessary vocabulary type.

But we do: `char32_t` models a Unicode code point. And sure, the original intent of `char32_t` is to represent a UTF-32 code unit, but there is no difference between a UTF-32 code unit and a code point, except that a code point is only 21 bits and UTF-32 code units are 0 extended to 32 bits.

The only motivation for a dedicated code point type would be to put a precondition of being in the range 0x00-0x10FFFF, but that adds a lot of complexity for a precondition that could be added elsewhere.

Similarly, we could entertain a more restrictive scalar value type, but then again Unicode algorithms are well-defined on any code points, including lone surrogates. Not producing lone surrogates is a post-condition of transcoding, not a precondition of algorithms.

And a scalar value type would not have any purpose besides maintaining that precondition.

Recommendation: We should use `char32_t` as the Unicode code point type

`charN_t` as code unit types

In the same vein, many previous works, including P2728R0 [6] introduce concepts and types to model UTF-8, UTF-16, and UTF-32. I think this is a mistake that adds gratuitous complexity given that C++ already has `char8_t`, `char16_t`, and `char32_t` whose only purpose is to model UTF-8, UTF-16 and UTF-32 respectively code unit types already.

We should use those in the Unicode algorithms interfaces directly, without abstractions. I do not want to open a debate about the usefulness of these types in the standard library and user-facing code, at least not here. There is a separate question of the I/O support for these types and their integration into existing facilities, etc.

But when transcoding from and to UTF-8, i.e. when manipulating sequences of UTF-8 code units, we should use the type we standardized, with great effort, for that purpose. This is very important, for two reasons.

Firstly, for simplicity. It's a recurring theme for me and these papers. We know that on average people find Unicode confusing. The more we can avoid introducing more types and concepts and streamline the interfaces, the less likely it is that these things will be used incorrectly.

Secondly, to force users to be explicit about intent. Even if the decoding process have the opportunity to report detectable errors (and not all decoding error are detectable), we should not assume that `char`, `intN_t`, `std::byte` and other integral types represent UTF data. As we know Unicode and text encodings are not necessarily well understood by developers, we should be simple…but explicit.

Allowing in UTF decoding interfaces and in Unicode algorithms interfaces arbitrary integral types is a recipe for disaster, as the type of arbitrary integral types do not carry Unicode semantics, and types like `char` can represent multiple categories of data.

At the same time, we know that *sometimes* `char` is used to represent UTF-8 data, and we do need to support that use case. But we should support that use case with an *explicit* interface.

P2626R0 [5] proposes a general solution to this problem, by allowing explicit casting from `char` to and from `char8_t` sequences. P2626R0 [5] does require some core language magic to work, but only because it strives to preserve the contiguous property of the resulting sequence of code units.

This is strictly unnecessary for Unicode algorithms (which cannot take advantage of contiguous sequences), and it would only be useful only for transcoding facilities that would require or benefit from contiguous sequences - something that would be only useful for SIMD transcoding, which again could not be used as an input of Unicode algorithms.

As such, for the purpose of Unicode algorithms and transcoding we only need a view that performs a `bit_cast` on each element. In my experience, that has a 0 runtime cost and provides some compile time safety.

Hold on: Why are we talking about code units if Unicode algorithms are code points transformations?

Good question. It's true that UTF-8, UTF-16, and UTF-32 are serialization formats and need to be deserialized to Unicode code points to be useful to Unicode algorithms, so in theory, we could separate entirely decoding and encoding from text processing.

Assuming an `uppercase` that takes a sequence of `char32_t`, we could do that:

```
std::u8string str = u8"Hello";
str = str | to_code_points |  views::uppercase | to_utf8 | to<std::u8string>();
```

And this would work perfectly. Each step is a different facility and the pipeline is very clear and explicit. There is a decoding step, a transformation powered by an algorithm, and then a re-encoding step to get back to our original type.

But...in practice, that would be a very cumbersome interface to use.

Saying that UTF-8 and UTF-16 are just serialization formats for code point sequences is entirely correct, but neglects to mention that Unicode data is almost always in either one of these formats, as storing code points as 32 bits values is most often a waste of memory, and so almost never done.

Forcing explicit decoding and encoding every time someone wants to uppercase a string would be extremely user-hostile, and the need for a friendlier interface would manifest immediately.

The code we'd like to write is more often something like that:

```
std::u8string str = u8"Hello";
str | views::uppercase | to<std::u8string>();
```

Let's just support UTF-8 then!?

Alas, some environments use UTF-16, some programs have UTF-32 data, and some have UTF-8 data. Sometimes, it's for historical reasons (which does not mean that reason is not valid or important), or for various technical reasons, which may be valid. The internet has spent 25 years arguing the benefit of one encoding over the other and I seriously doubt anyone in WG21 could bring anything new to the table.

The arguably sad reality is that C++, **if we want to be useful to most users, probably can't afford to be opinionated here, and need to support UTF-8, UTF-16, and UTF-32 alike**.

Can we just... not?

At the same time, I don't think the idea of supporting multiple encodings in all algorithms is appealing at all. Forcing every Unicode algorithm/view to be, in addition, a UTF-decoder/encoder adds complexity both in the spec, in implementations, and for users. We could also create performance pitfalls or unexpected behaviors when chaining algorithms.

Transcoding brings additional complexity compared to Unicode algorithms operating on code points: namely, transcoding can fail whereas Unicode algorithms usually can't.

So on one hand we would want to support all UTF forms for the sake of ergonomy and at the same time the duplication in triplicate of all algorithms + having every Unicode facility in the standard be a transcoding facility does not seem to be a great proposition.

Eating our cake and having it too: Range adaptors & pipe operators

In practice, views are more often use trough range adaptors (pipe syntax), which offers a more ergonomic and readable interface (because of the natural reading order, the conversion of range to views, and the support for partial applications).

And if we are going to provide views for Unicode algorithms, we are going to provide range adaptors for these views. And here is the key insight: we can add support for different UTF encodings in the range adaptors, and only there.

The idea is that

```
auto str = as_utf8("Hello") | views::uppercase | ranges::to<std::u8string>();
```

is equivalent to

```
auto str = utf8_view(uppercase_view(codepoint_view(views::all(as_utf8("Hello")))))
 | ranges::to<std::u8string>();
```

That is to say, the range adaptor pipe operator has overloads for `charN_t` that insert decoding and encoding views on either side of the Unicode algorithm.

This has several benefits. For the user, it's entirely transparent and expresses the intent of applying a Unicode algorithm, they don't have to care or be explicit about encodings, code units, and code points if they don't want to (and again, as most people needing to perform text transformation are blissfully(?) unaware of the intricacies of Unicode, this is a desirable property).

For the specification, implementers, and users, it keeps the complexity of encoding in a single place and does not force us to break abstraction barriers for the sake of ergonomy (and indeed, the code of these range adaptors can be reused).

Avoiding redundant work

Let us look at a more complicated example.

```
auto str = u8"Hello" | views::normalize | views::uppercase | ranges::to<std::u8string>();
```

Naively this is equivalent to:

```
auto str = utf8_view(uppercase_view(codepoint_view(
           utf8_view(normalize_view(codepoint_view(u8"Hello"))))))
          | ranges::to<std::u8string>();
```

Indeed, if we assume that:

1. Users want to implicitly get back the same encoding as output as they provided as input
2. Unicode algorithms are often composed

assumptions that from experience are very fair assumptions to make, then a naive approach to composition would create a lot of intermediary encode/decode steps that have no effect besides burning CPU cycles.

Unsurprisingly, Zach Laine came to the same conclusion, and the solution proposed in his work is to let iterators peek through layers of encode/decode iterators through their `base()` function - (such function exists on views and views' iterators of the `<ranges>` design and return an underlying iterator).

The conclusion I came to is that we can let range adaptors add or remove these implicit decode functions when they are chained before any view is constructed. The effect is the same, although the implementation is simpler, as maintaining a `base()` iterator can be tricky, especially in the presence of a non-common or reverse iterator.

Tailoring again

Earlier, I explained that we might want tailored algorithms to offer different guarantees and interfaces from non-tailored algorithms, as tailored algorithms would have to rely on third-party libraries and runtime data, all of which can throw, cannot be constexpr, could allocate, may not satisfy the bidi requirements, and so forth (and at a minimum, the tailored version needs to store a local identifier object of some kind which, surprise, would be an ABI break if done later!).

Range adaptors are again a solution to these constraints.

Indeed, we can envision 2 sets of views, for example

```
class default_uppercase_view {
    default_uppercase_view(view_of_char32_t);
};
class tailored_uppercase_view {
    uppercase_view(std::ulocale, view_of_char32_t);
};
```

and a single range adaptor `uppercase` that would construct either view depending on whether it is called with a locale parameter or not.

```
auto str = u8"Hello istanbul";
str | views::uppercase // utf8_view(uppercase_view(code point_view))
str | views::uppercase("tr") // utf8_view(tailored_uppercase_view(code point_view))
```

I believe this would be the best, if not the only way to have consistent interfaces for tailored and non-tailored APIs, without breaking ABI if locale is not fleshed out in the 26 cycle (which is unlikely, to say the least).

Implementation and Performance considerations

I think the first question we have to contend with is: ICU or not ICU. IE, do we design the interface in such a way that it can be implemented by reusing existing third parties libraries. This will constrain the design, quite radically.

The libraries to consider are mostly ICU and ICU4x. Some operating systems (OSX, windows, some GNU systems through `libunistring`) do provide some Unicode algorithms. However, we won't find a set of requirements that could be satisfied by most of these libraries. For example, the set of algorithms for which ICU, win32 and other expose iterators is...limited at best.

I did not have the time to investigate the capabilities of these libraries, or their license (a question best answered by implementers anyway), but we would be seriously limited if we tried to design Unicode in a way that satisfies the common subset of capabilities of the underlying platforms we want to support.

And while it may not be reasonable to expect tailored collation to work on a toaster, some algorithms, in particular normalization and casing, should probably be supported on embedded systems and ICU is certainly not going to run there.

Both I and Zach Laine have implemented Unicode algorithms with no external dependencies. I focused on non-tailored algorithms while Zach went as far as implementing collation (which is quite a feat).

My own experimentations make a point for all non-tailored algorithms to

- Provide bidirectional iterators (through views)
- Be constexpr
- Be noexcept

This ends up having many benefits:

- The interface of all algorithms, and their requirements are consistent, all of these things have the same shape and API. As they satisfy all the range requirements they compose with everything else. they are just more views, albeit domain specific. But should you want to zip a reverse view of graphemes, it's certainly possible.
- It's 100% portable: As the algorithms have no dependencies, they run on everything from a toaster to the TOP500 supercomputers, and the zOS of the world, which is more or less a requirement of standard libraries component, especially for implementations that avoid dependencies to ease distribution.
- Consistent behavior across platforms, faster adoption of newer Unicode versions in the toolchain
- Implementers can reuse these components for the work on source code spoofing in compilers, which I suspect they will want to look at in the coming year.
- We can support more use cases such as CTRE, constexpr fmt, etc
- Possibility to only link the Unicode tables that are actually used by the application, which allows more devices to support Unicode (say, case insensitive search on a cheap smartwatch)
- Better performance through inlining

At the very least, for things for which we do think implementers will want to rely on a third party, we should not focus on ICU as much as ICU4x as this is where most of the efforts in terms of portability, performance, and conformance are being invested currently. My hope is that future efforts for tailored algorithms and locale-dependent features in C++ are designed with ICU4x in mind.

Implementing non-tailored algorithms without external dependencies

As mentioned previously, most Unicode algorithms operate by looking at code point properties and replacing or altering code point sequences in some way based on these properties. In my experience, 90% or more of the work of implementing Unicode algorithms is implementing these property lookups.

Indeed, different look-up tables implementations are most suited to given properties. Not only do properties have different types (boolean, a finite amount of state, individual or sequence of code points, etc), but they have varying spareness that can be exploited.

A lot of work has been done (by Zach, the Rust community, myself, and others) recently to explore different strategies for storing these properties.

This is absolutely critical, as the performance we can offer is directly tied to that of these lookups. Normalization has some subtleties, collation and the bidi algorithms are even more so but clusterization and casing are simple and straightforward. I don't even think there are major difficulties but creating and maintaining these tables is somewhat time-consuming.

Generally speaking, reducing the size of the tables has a direct impact on performance, if only because increasing cache locality is the most effective way to improve the performance of anything.

I landed on a set of strategies developed by the rust team, in addition to using perfect hashing for combining classes, and realizing that the standard `binary search algorithms` can be optimized, for the specific data used by Unicode algorithms. I also explored, with great success implementing SIMD-based lookups for the data structures developed by the Rust team.

The question I keep asking myself is... can we somehow share that work across implementations? While I do believe that the standard would benefit from portable Unicode algorithms, it's not clear to me standard libraries implementers have the bandwidth to maintain competitive implementations.

At the very least, we should offer some guidance to maximize the change of high-quality implementations.

Unicode algorithms are not sized. Really?

All Unicode algorithms are $N->M$ transformations. I've said that before, I'm a broken record. But it's important. One because, as we saw earlier there is no point in trying to do in-place Unicode transformations, but also because of the performance implications.

```
long_utf32_string | views::uppercase | to<std::u32string>();
```

This nice-looking code has the potential to have terrible performance. It's not as bad as trying to do inserts-in-the-middle as the string is transformed, but it's still bad because `ranges::to` cannot reserve memory in advance. We can't possibly know the size of the output before the end of the transformation, so that code will be equivalent to a series of `push_back`. At best the container will have a reasonable growth factor, but still, that's where the performance pitfall is, in those allocations. As far as I was able to observe (I still need to run more benchmarks), these allocations (or rather constant resizing and memory shuffling) can dwarf everything else the algorithm may be doing.

The thing is… Unicode algorithms are statistically $N- > N$ transformations, with some edge cases. For most code points their casing transformation is the identity functions, most code points are unchanged by normalization, and most graphemes are of size 1. Of course, this depends on the script, so different content will have different behaviors, but…most of the time the output size will be the input size or very close to that.

This is something we can exploit.

`ranges::to` is missing a size hint, to help it reserve memory for non-size ranges. It has nothing to do with Unicode, in that it something SG-9 and LEWG can pursue independently, but at the same time, I cannot think of a more compelling use case.

Rust has such utility (Rust later found that having a pair of bounds is not actually useful, a single value is more usable), and it's something we can bring to C++ and that would have a great impact on performance for Unicode algorithms.

I plan to work on that shortly.

Algorithm-specific considerations

Specificities of normalization

One of the operations done by normalization is to sort combining code points according to their class. As such an implementation must maintain an internal buffer of code points. Alas, sequences of combining code points are unbounded, so normalization algorithms can be forced to allocate to grow that buffer.

But. Long diacritics sequences while technically possible do not appear in normal text (we get into zalgo territory), and there is a "safe stream" specification to limit the sequences of combining code points to something more than large enought to account for all natural languages.

While that spec was developed for safety reasons (this peculiarity can be exploited and turned into a denial of service), it also allows a non-allocating normalization implementation.

Where I'd diverge from P2729R0 [7] is how we expose these different options. P2729R0 [7] exposes the notion of a safe stream directly.

However, this forces the user to know that this exists and that the default can allocate, and the normalized algorithms/views need to be "safe stream aware" to be able to not allocate. It all seems to me awfully complicated for all involved, especially as this notion has little use outside of normalization

My favorite solution to handle that would be for normalization to use the safe stream algorithm by default, and to have that behavior be overridable by letting the user pass a buffer of some kind (likely a polymorphic resource), which has the benefit of being efficient by default and putting on users the responsibility to handle allocation failures.

The other specificity of normalization, as noted in P2729R0 [7] is that concatenating or substituting in a normalized string only has to modify the sequences on either side of the insertion/removal point.

I'm not sure there is a better way to do that compared to P2729R0 [7], it ends up being quite a bit of surface area. I'm not sure it's worth focusing on too much in the first set of features, the strings have to be pretty big or numerous for the cost of renormalizing everything to be observable. It's certainly a nice set of features in the long run.

Specificities of grapheme clusterization

There is nothing very complicated about grapheme clusterization, it's just a group_by view, the kind of thing LEWG eats for breakfast. Except that hopefully, grapheme clusterization produces graphemes.

How to model graphemes is still something that I'm exploring. Zach and I landed on the same conclusion: we need some kind of `grapheme_ref` type (basically `span<char32_t>`) and a `grapheme` type, that owns a sequence of `char32_t`. Ideally, I would have wanted for `grapheme_ref` and `grapheme` to be the reference and value types of grapheme ranges respectively.

Alas, my recent difficulties around P2165R4 [4] make me think that this might not be possible. Which would be a shame. Should `graphemes | to<std::vector>()` produce a range of dangling views, it would be a problematic shotgun.

I also struggle to find the ideal layout for these things. Graphemes are on average one code point and are numerous. There is value in reducing their memory footprint. For example, 64 bits is enough to store 3 code points + the additional machinery to allocate space for more if needed.

I think the implementation of graphemes can be simple as they don't need to support most container operations like erasure and insertion. But they are still containers.

`grapheme_ref` and `grapheme` are important vocabulary types. In particular, future Unicode properties interfaces (P1628R0 [2]) need to support them, in addition to code points, as graphemes do have properties.

Opinions on UTF decoding

UTF decoding is not particularly interesting to me but it is certainly very foundational so, like probably most of SG-16, I've implemented many iterations of it, which led me to have opinions on the matter.

- By default, we must assume that UTF-sequences may be ill-formed, and failing to check this assumption introduces vulnerabilities.

- For performance reasons, there is a need for non-checking, opt-in interfaces that are very explicit about their potential to tear down limbs. Nevertheless, in a closed system that produces well-formed input, this is necessary.
- UTF encoding and decoding are very fast operations. We could spend years chasing theoretical zeros (an exercise implementers may not be inclined to follow us in), and support SIMD, etc, but the very fast interfaces are not very usable (there has not been much progress in decades on iterator interfaces for SIMD), and these supposedly ultra performant interfaces often perform terribly on short or isolated strings, as they require a lot of initial state or CPU warm up.
- We could have multiple interfaces (like P2728R0 [6] which has both an eager and a ”convenient” interface), but we should do so with the understanding that for the very few use cases that do require a very performant UTF transcoding, implementers are unlikely to be able to compete with the likes of Daniel Lemire.
- Eager very fast decoding does not compose with Unicode algorithms, the same way that eager algorithms never compose with anything.
- For the purpose of producing a bidirectional view of code points, the most performant and flexible approaches are The use of a small finite automaton, or similar techniques.

Ultimately, the ”I have lots of small UTF strings in my UI” and ”I'm trying to decode this 10GB JSON file representing a language model” are very different use cases and cannot be served by the same interface.

The real question for me is, who wants to eagerly decode 40GB of code points?

Note that I would love to see research and experimentations with range-based decoding that would use SIMD, satisfy the range requirements, and preserve the `base()` iterator, but I'm not sure this work will happen or be actionable.

Maybe we don't need to decode at all?

We talked many times about the idea of not needing to decode at all, and just to do the whole Unicode transformations, including lookup and replacement as UTF-8 or UTF-16 code units.

But... does that work?

I think this overestimate the cost of decoding, and underestimate the cost of everything else. A naive UTF-8 decoding is *fast*, or at least it is relatively fast compared to say normalization.

But mostly... even if we could not decode the UTF-8 input, we'd still have to validate it, and subsequently handle errors. We'd had to find a way to look up UTF-8 sequences (of variable length), which constrains the kind of implementation strategies we can use for Unicode, a lot.

Or we would have to stitch together a bunch of code units in a 32 bits integer, a process that would be almost as expensive as decoding unless the underlying stream is contiguous, maybe.

Then there is the matter of UTF-8 wasting a bunch of bits (that are used to identify individual code units), so a code unit sequence might use 32 bits whereas a code point only uses 21, leaving 11 bits to store properties or look up metadata. So we might have to shift and mask our UTF-8 sequence... which would be more expensive than decoding.

And on the output side, we'd have to store our replacement code points as variable UTF-8 code points, which is inefficient, either CPU or memory wise, again to avoid UTF-32 to UTF-8 encode, an operation that is much faster than most property lookups.

It gets worse. By having property tables per encoding, we not only waste a bunch of memory, but we risk thrashing CPU caches for people who intentionally or accidentally mix encodings.

And lastly... would implementers, if offered, be willing to implement algorithms in triplicate, when we are still debating whether they will be inclined to do it once?

Finding stability when there is none: Keep the interfaces opaque

Unicode algorithms can have behavior changes from one Unicode version to the next, both because the algorithm itself can change and because code point properties not explicitly documented as stable can change.

The best way to protect ourselves against surprises is to avoid exposing any internals or assumptions about the algorithms.

The transformations are best left as a black box. We should not give users the opportunity to perform their own tailoring, for example, as this would force us to design interfaces that make too many assumptions about the capabilities of tailoring, or the semantics of properties.

Both ICU and ICU4x have a similar "service" model, where the locale identifier has sole control over the behavior of algorithms which are then not user customizable.

In the same vein (and perhaps in contradiction of my previous work on P1682R0 [8]), I think we should be careful not to expose properties that are specific to a Unicode algorithm, and have no other use, on a case by case basis.

There are two reasons for that: One the implementation of the algorithm may use an implementation strategy that would not be well-suited for a property look up interface, and because a user implementation of the algorithm is unlikely to use the standard look up interface (because why would you reimplement an algorithm if lookup is already optimally efficient?).

That does not mean no property should be exposed. For example, we may consider exposing the simple casing properties in addition to the full-casing algorithm.

The one use case, that does requires, depending the level of conformance exposing most properties is Unicode regexes, but then again, it's likely an implementation may want to deviate from whatever look up interface the standard provides.

Implementation experience

My resources being scarce and my time limited, I unfortunately do not have an implementation nearly as mature as Zach Laine's.

Nevertheless, I've over the past years gone through multiple iterations of some of these interfaces, so I'm starting to have a pretty good understanding of the amount of work needed and the general shape of the interfaces we need. In particular, I've shown that a conforming reimplementation of normalization can be faster than ICU's.

That work (very much a prototype that may not compile on anyone's) machine but my own) is available on Github.

I hope to find the time to continue that work.

References

[1] Tom Honermann. P0244R2: Text_view: A c++ concepts and range based character encoding and code point enumeration library. `https://wg21.link/p0244r2`, 6 2017.

[2] Corentin Jabot. P1628R0: Unicode characters properties. `https://wg21.link/p1628r0`, 6 2019.

[3] Corentin Jabot. P2020R0: Locales, encodings and unicode. `https://wg21.link/p2020r0`, 1 2020.

[4] Corentin Jabot. P2165R4: Compatibility between tuple, pair and tuple-like objects. `https://wg21.link/p2165r4`, 7 2022.

[5] Corentin Jabot. P2626R0: charn_t incremental adoption: Casting pointers of utf character types. `https://wg21.link/p2626r0`, 8 2022.

[6] Zach Laine. P2728R0: Unicode in the library, part 1: Utf transcoding. `https://wg21.link/p2728r0`, 12 2022.

[7] Zach Laine. P2729R0: Unicode in the library, part 2: Normalization. `https://wg21.link/p2729r0`, 12 2022.

[8] JeanHeyd Meneide. P1682R0: std::to_underlying. `https://wg21.link/p1682r0`, 6 2019.

[N4892] Thomas Köppe *Working Draft, Standard for Programming Language C++* `https://wg21.link/N4892`