Boost.Text: Fixing `std::string`, and Adding Unicode to Standard C++

Zach Laine, C++Now 2018

How did we get here?

Alisdair Meredith is evil.

SG-16 are evil.

Part 1: Motivation

`std::string` sucks

- It has a fat interface, full of member functions that should have been algorithms.

```
// Whiskey tango foxtrot
auto index = std::string("something to search").find_last_not_of(" search");
```

- It's badly suited to large, frequently-edited strings
- There is no text encoding support

Standard C++ has low-to-no general-purpose Unicode support.

This is a bit embarrssing in 2018.

Library Goals

Replace the full functionality of `std::string`, doing things the right way.

- The new `string` has an expressive and efficient basis
- Many member functions become proper algorithms
- Everything is range-friendly
- Sub-stringing should be easy and efficient
- The `size_type` should be signed
- Ridiculously large strings are not a reasonable use case, so the `size_type` should be 32 bits

Add all the missing stuff.

- Support COW and SBO optimizations
- Support thread-safe strings
- Support efficiently-editable large strings
- Robust Unicode support
- The Unicode bits should be easy to use for those not familiar with Unicode

Don't require the use of Unicode and/or encoded text for users that don't need it.

Target standardization.

Support every standard from C++11 on, in order to gather the most feedback.

Radical simplicity. Even new users should be able to grok string and text processing.

- Use as few templates as possible
- Overload sets should be as small as possible
- No allocators
- Add new functionality only when the radically simple approach proves unworkable

Part 2: Introducing Boost.Text

Boost.Text is in designed around three layers.

- The `string` layer. This layer knows nothing about the other two layers.
- The Unicode layer. This layer has optional interfaces that use some of the `string` layer (specifically, `text::string`), but knows nothing about the `text` layer.
- The `text` layer. This layer depends on the other two.

Note that the `string` and Unicode layers may be used as standalone code without using the other layers.

The `string` layer.

- `text::string`
- `text::string_view`
- `text::unencoded_rope`
- `text::unencoded_rope_view`
- `text::repeated_string_view`

text::string

```
struct string
{
    using iterator = char *;
    using const_iterator = char const *;
    using reverse_iterator = detail::reverse_char_iterator;
    using const_reverse_iterator = detail::const_reverse_char_iterator;

    string() noexcept;
    string(...);

    ~string();

    string & operator=(...);

    iterator begin() noexcept;
    iterator end() noexcept;
    // etc.

    bool empty() const noexcept;
    int size() const noexcept;
    int capacity() const noexcept;
```

```
    char operator[](int i) const noexcept;
    char & operator[](int i) noexcept;

    string_view operator()(int lo, int hi) const;
    string_view operator()(int cut) const;

    int max_size() const noexcept;
    int compare(string_view rhs) const noexcept;
    void clear() noexcept;

    iterator insert(iterator at, ...);
    string & insert(int at, ...);
    string & erase(string_view sv) noexcept;
    string & replace(string_view old_substr, ...);

    void resize(int new_size, char c);
    void reserve(int new_size);
    void shrink_to_fit();
    void swap(string & rhs) noexcept;

    string & operator+=(...);
};
```

`operator+`, `operator<<`, and the equality and relational operators are also defined.

All the operations fit on two slides. There are no algorithms posing as member functions.

`text::string` can be constructed from any of the following:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer
- Any range of graphemes built on an underlying range of `char` modeling `GraphemeRange` (e.g. types from the `text` layer)

All constructors are `explicit`, except the one taking `char const *`.

Assignment is defined for the same types as construction.

`text::string` has these sub-stringing ("slicing") operations:

```
string_view operator()(int lo, int hi) const;
string_view operator()(int cut) const;
```

Our signed `size_type` makes it convenient to use negative indexing.

```
text::string const s("some text");

assert(s(2, 9) == "me text"); // slice
assert(s(2, -1) == "me tex");
assert(s(4) == "some");        // prefix
assert(s(-4) == "text");       // suffix
```

The slicing operations never allocate.

There is no allocating sub-stringing API.

`insert()` is defined for `int` indices and `iterator` positions, and accepts most of the same types that the constructors and assignment operators take:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer
- An arbitray `[first, last)` `CharIter` iterator pair

Note that types from the `text` layer are not supported for `insert()`. If they were, you may silently drop encoding and/or normalization.

The iterator interface can still be used with the `text` layer types. This forces the user to write more verbose code to opt in to dropping the encoding and normalization invariants.

Incoming text passed to `replace()` and `operator+=` is handled the same way -- `text` layer types are not supported. Explicit uses of more-verbose iterator interfaces is required.

Equality and relational operators are only defined between `text::string` and these types:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer

Again, comparison to `text` layer types is not allowed with a simplified syntax, because `string` layer types know nothing about encoding, normalization, or collation.

`std::equal()` and `std::lexicographical_compare()` of course still exist.

text::string_view

```
struct string_view
{
    using iterator = char const *;
    using const_iterator = char const *;
    using reverse_iterator = detail::const_reverse_char_iterator;
    using const_reverse_iterator = detail::const_reverse_char_iterator;

    constexpr string_view() noexcept;
    BOOST_TEXT_CXX14_CONSTEXPR string_view(...);
    BOOST_TEXT_CXX14_CONSTEXPR string_view(..., int lo, int hi);

    BOOST_TEXT_CXX14_CONSTEXPR string_view & operator=(...) noexcept;

    constexpr const_iterator begin() const noexcept;
    constexpr const_iterator end() const noexcept;
    // etc.
```

```cpp
    constexpr bool empty() const noexcept;
    constexpr int size() const noexcept;

    BOOST_TEXT_CXX14_CONSTEXPR char operator[](int i) const noexcept;

    BOOST_TEXT_CXX14_CONSTEXPR string_view operator()(int lo, int hi) const;
    BOOST_TEXT_CXX14_CONSTEXPR string_view operator()(int cut) const;

    BOOST_TEXT_CXX14_CONSTEXPR int compare(string_view rhs) const noexcept;
    BOOST_TEXT_CXX14_CONSTEXPR void swap(string_view & rhs) noexcept;
};
```

`operator<<`, and the equality and relational operators are also defined.

`text::string_view` can be constructed from any of the following:

- `char const *`
- Any range of `char` modeling `ContigCharRange` (e.g. `std::string`)
- Contiguous-storage types from the `string` layer
- Any range of graphemes built on an underlying range of `char` modeling `ContigGraphemeRange` (e.g. contiguous-storage types from the `text` layer)

All constructors are non-`explicit`, except the generic ones accepting `ContigCharRange` and `ContigGraphemeRange` types.

Assignment is defined for the same types as construction.

Slicing operations are the same as for `text::string`, and do not allocate:

```
text::string_view const s("some text");

assert(s(2, 9) == "me text"); // slice
assert(s(2, -1) == "me tex");
assert(s(4) == "some");       // prefix
assert(s(-4) == "text");      // suffix
```

Equality and relational operators are only defined between `text::string_views`. The implicit conversions of many types to `text::string_view` implies comparison to these types:

- `char const *`
- Contiguous-storage types from the `string` layer

Comparison to `text` layer types is not allowed.

text::unencoded_rope

Juan Pedro Bolivar Puente is evil.

`text::unencoded_rope` is a tree-based data structure based on a B-tree. Interior nodes exist for structure, and leaf nodes contain string data.

Each leaf node is one of:

- `text::string`
- `text::repeated_string_view`
- A reference to a slice of a `text::string` leaf node

Each interior node's child-index array occupies exactly two 64-byte cache lines.

Each node (and thus subtree) is pointed to via a reference-counted COW pointer. Copying a `text::unencoded_rope` only requires copying a pointer and incrementing an atomic ref-count; the copy shares all the nodes from the original.

In a typical COW implementation, mutating a COW object entails first copying it, then mutating the copy.

If you edit a `text::unencoded_rope` such that the edit would fall entirely with a node `N`, and the path from the root to `N` contains only nodes with a reference count of `1`, `N` is mutated in place.

Otherwise, `log-B(N)` new nodes must be created (one for each node from the root to `N`), and the other nodes are shared between `*this` and the other `text::unencoded_ropes` pointing to the rest of the nodes.

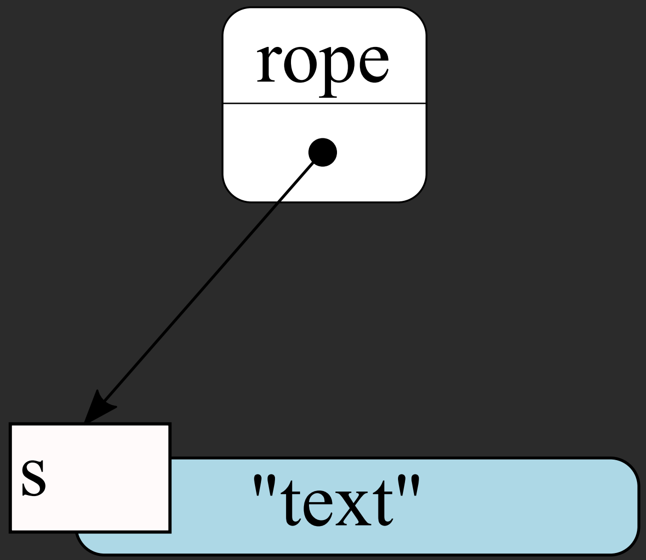
The branching factor `B` is between 8 and `16` in the current implementation.

This puts an upper bound on `log-B(N)` of `16`, given the precision of the signed 64-bit value used for `size_type`. This implies that `log-B(N)` can be treated as a constant factor, allowing efficient random access to elements in `text::unencoded_rope`.

Pretty pictures!

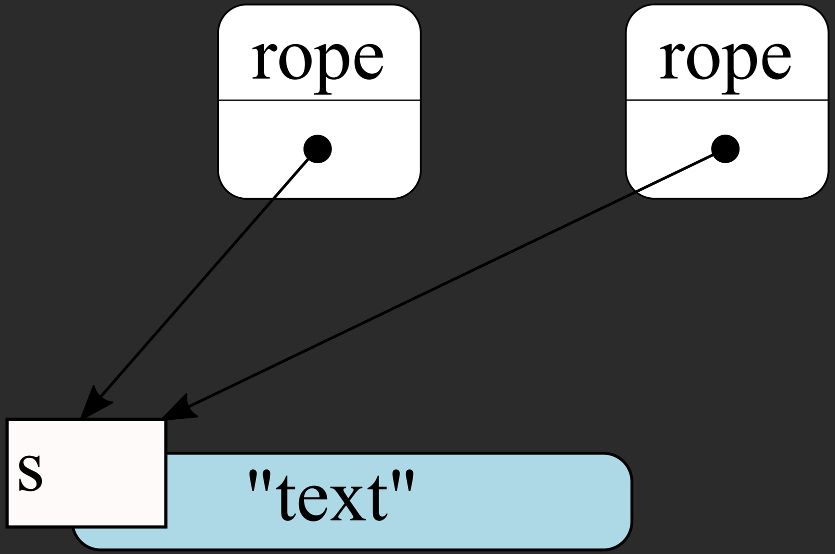
In these diagrams, `s`, `rsv`, and `ref` are used to refer to `text::string`, `text::repeated_string_view`, and `text::string`-reference nodes, respectively.

Here is one of the simplest non-empty `text::unencoded_ropes` you can have:



This `text::unencoded_rope` has only a single `text::string` leaf node.

If we make a copy, we get this:

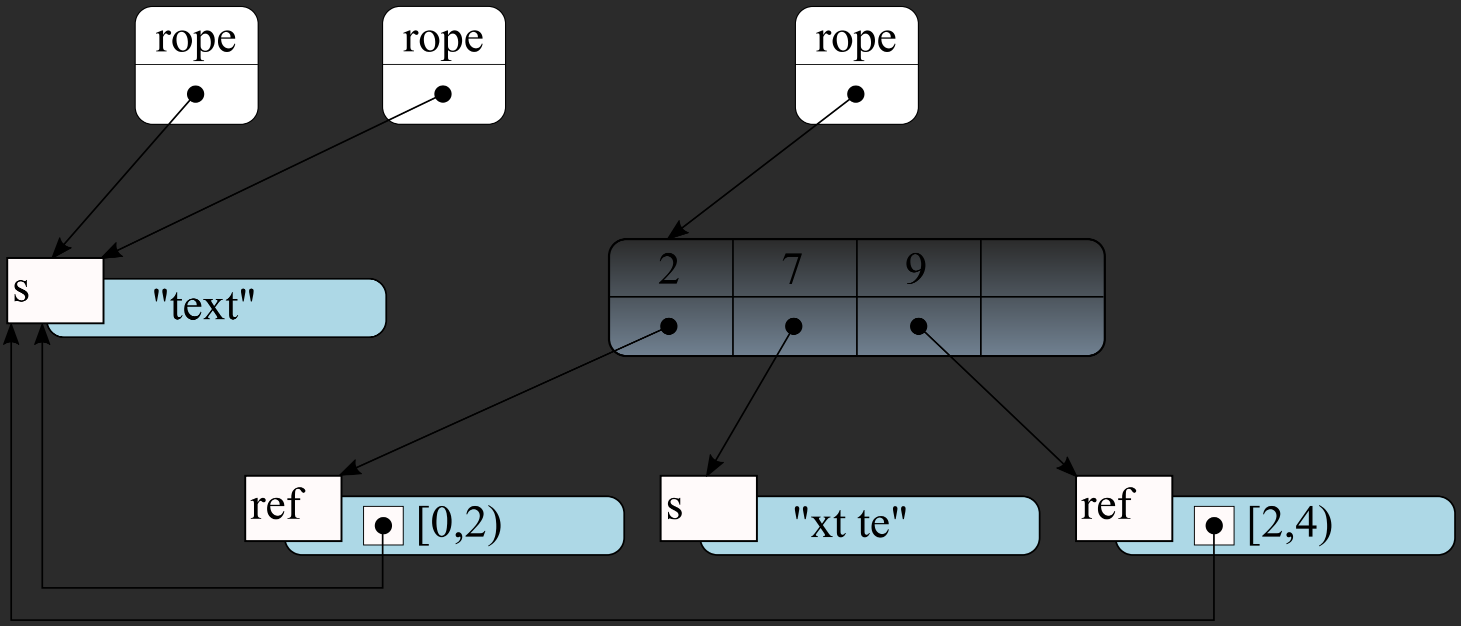


That's two `text::unencoded_rope`, each of which is sharing that one leaf node.

No copying was done (besides the root pointer); no allocations were necessary.

If "`text`" were instead a large tree with a gigabyte of characters in it, the copy would cost the same.
This makes undo systems trivial to write -- just use a stack of `text::unencoded_ropes`.

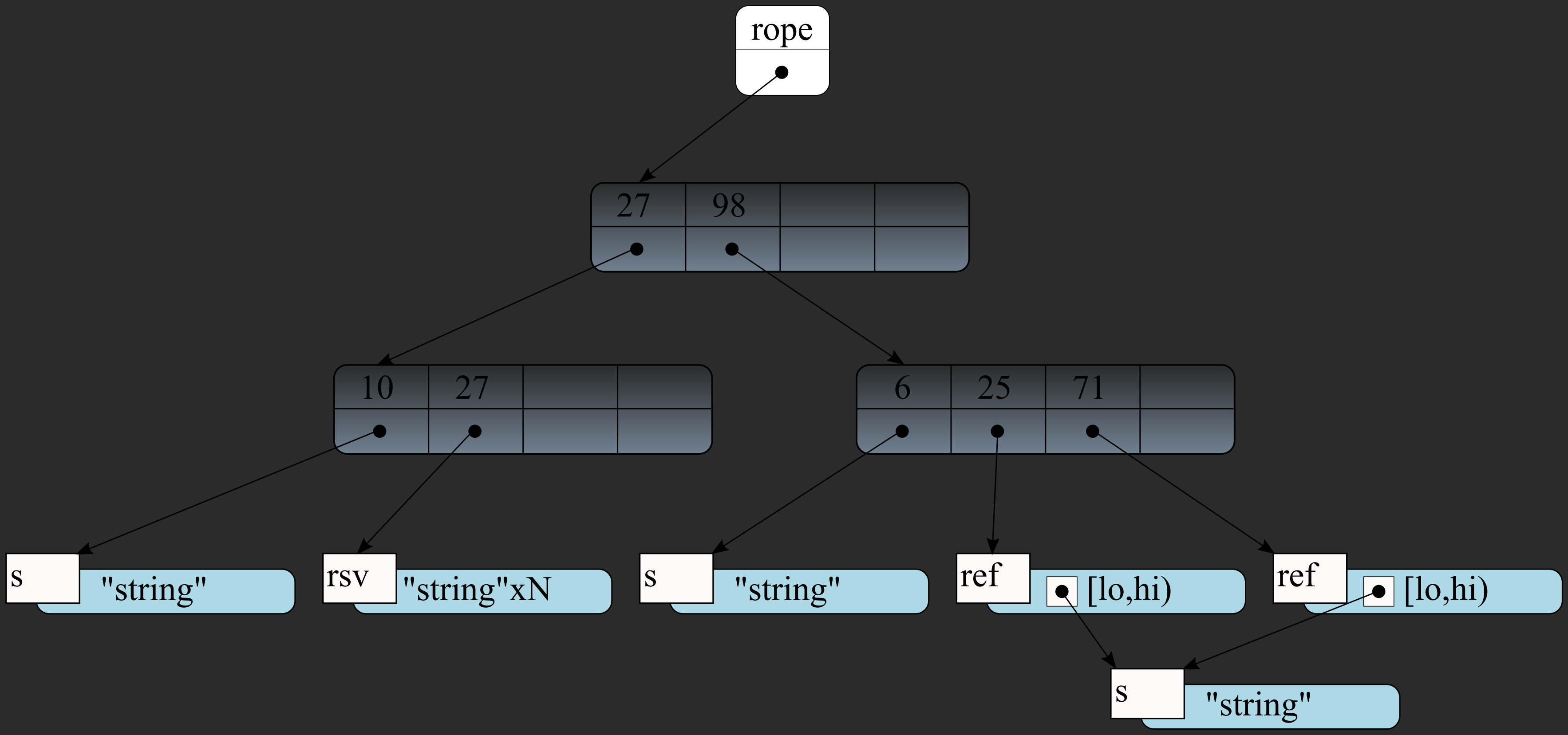
Let's say we want to insert some text into the middle of "`text`", to form the new string "`text text`":



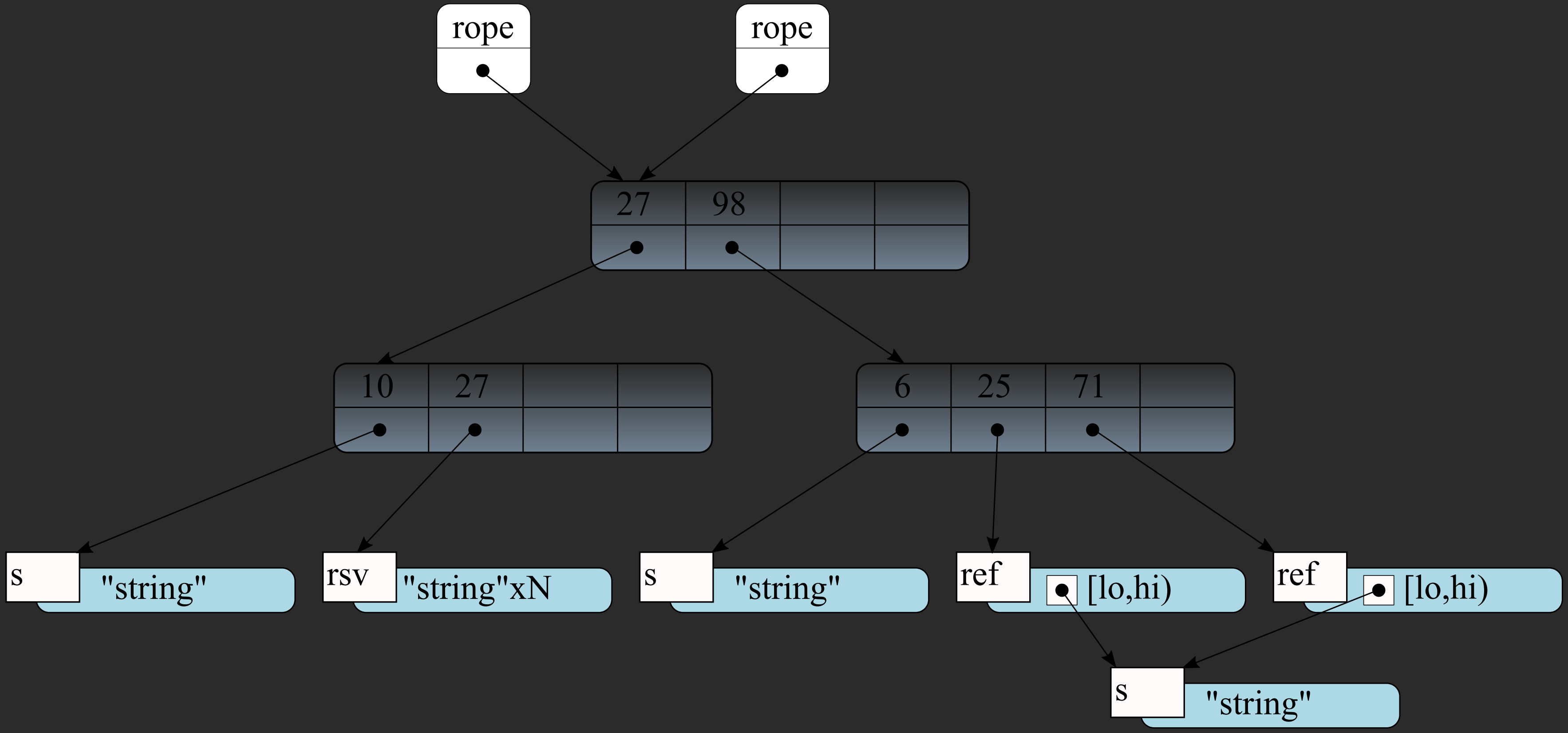
The result is that the original tree is unchanged, and we now have a new one that refers to parts of the original.

With our rope containing only `"text"`, this is not a big deal. Imagine if `"text"` were instead a megabyte.

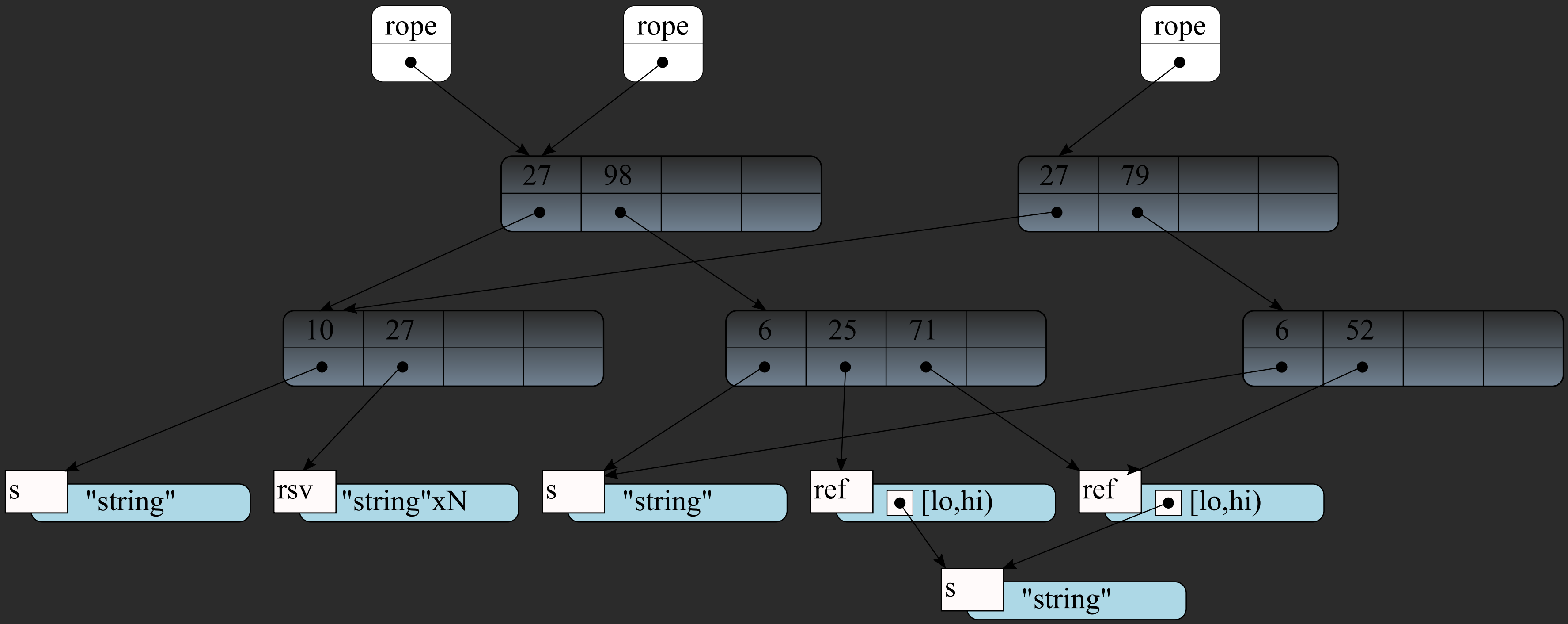
A more complicated `text::unencoded_rope`:



Copying it is cheap, as we saw before:



How about if we erase the leftmost string reference of the `text::unencoded_rope`?



So simple!

In this case, the erasure by created a copy of each node on the path from the root to the erased leaf, and just referred to all the other nodes that did not change.

Most of the string data and even interior nodes are shared among the three ropes in the diagram. This same principle applies to `insert()`, `erase()`, and `replace()`.

As long as you always pass `text::unencoded_rope` by value (which is cheap), any use of it is thread-safe.

text::unencoded_rope

```cpp
struct unencoded_rope
{
    using iterator = detail::const_rope_iterator;
    using const_iterator = detail::const_rope_iterator;
    using reverse_iterator = detail::const_reverse_rope_iterator;
    using const_reverse_iterator = detail::const_reverse_rope_iterator;

    unencoded_rope() noexcept;
    unencoded_rope(...);

    unencoded_rope & operator=(...);

    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    // etc.

    bool empty() const noexcept;
    size_type size() const noexcept;

    char operator[](size_type n) const noexcept;
```

```
    unencoded_rope_view operator()(size_type lo, size_type hi) const;
    unencoded_rope_view operator()(size_type cut) const;

    size_type max_size() const noexcept;

    unencoded_rope substr(size_type lo, size_type hi) const;
    unencoded_rope substr(size_type cut) const;

    template<typename Fn>
    void foreach_segment(Fn && f) const;

    int compare(unencoded_rope rhs) const noexcept;
    bool equal_root(unencoded_rope rhs) const noexcept;
    void clear();
    void swap(unencoded_rope & rhs);

    unencoded_rope & insert(size_type at, ...);
    const_iterator insert(const_iterator at, ...);
    unencoded_rope & erase(unencoded_rope_view rv);
    unencoded_rope & replace(unencoded_rope_view old_substr, ...);

    unencoded_rope & operator+=(...);
};
```

`operator+`, `operator<<`, and the equality and relational operators are also defined.

`text::unencoded_rope` can be constructed from any of the following:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer
- Any range of graphemes built on an underlying range of `char` modeling `GraphemeRange` (e.g. types from the `text` layer)

All constructors are `explicit`, except the one taking `char const *`.

Assignment is defined for the same types as construction.

Slicing operations are the same as for `text::string` and `text::string_view`, except that they return an `text::unencoded_rope_view`. They do not allocate.

```
text::unencoded_rope const s("some text");

assert(s(2, 9) == "me text"); // slice
assert(s(2, -1) == "me tex");
assert(s(4) == "some");        // prefix
assert(s(-4) == "text");       // suffix
```

```
template<typename Fn> void foreach_segment(Fn && f) const;
```

Since a `text::unencoded_rope` is broken up into segments, some operations may benefit from operating on one segment at a time:

```
auto f = [](auto && s) {
    for (auto c : s) {
        // X
    }
};

text::unencoded_rope const s(/*...*/);
s.foreach_segment(f);
```

If the operation X is very simple, not having to branch on every iteration is probably faster.

The remaining operations follow the same rules as `text::string` with respect to which types the operate on:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer
- `text` layer types are supported, but only with a more verbose interface

The COW properties of `text::unencoded_rope` allow writing extremely simple and efficient undo systems:

```
std::vector<text::unencoded_rope>::iterator
new_undo_state(std::vector<text::unencoded_rope> & history,
               std::vector<text::unencoded_rope>::iterator it)
{
    history.erase(std::next(it), history.end());
    return history.insert(history.end(), history.back());
}

std::vector<text::unencoded_rope>::iterator
insert(std::ptrdiff_t at,
       text::unencoded_rope_view rv,
       std::vector<text::unencoded_rope> & history,
       std::vector<text::unencoded_rope>::iterator it)
{
    it = new_undo_state(history, it);
    it->insert(at, rv);
    return it;
}

// Maybe add erase(), replace(), etc.
```

```cpp
std::vector<text::unencoded_rope>::iterator
undo(std::vector<text::unencoded_rope> const & history,
     std::vector<text::unencoded_rope>::iterator it)
{
    if (it != history.begin())
        --it;
    return it;
}

std::vector<text::unencoded_rope>::iterator
redo(std::vector<text::unencoded_rope> const & history,
     std::vector<text::unencoded_rope>::iterator it)
{
    if (std::next(it) != history.end())
        ++it;
    return it;
}
```

```cpp
std::vector<text::unencoded_rope> undo_history(1);
std::vector<text::unencoded_rope>::iterator current_history =
    undo_history.begin();

// Copies the initial empty rope; inserts into the copy; returns a pointer to
// the result.
current_history = insert(0, "Yay!", undo_history, current_history);

// Conditionally decrements a pointer (pointing back to the initial empty
// rope).
current_history = undo();

// Conditionally increments a pointer (pointing back to "Yay!").
current_history = redo();
```

text::unencoded_rope_view

```
struct unencoded_rope_view
{
    using iterator = detail::const_rope_view_iterator;
    using const_iterator = detail::const_rope_view_iterator;
    using reverse_iterator = detail::const_reverse_rope_view_iterator;
    using const_reverse_iterator = detail::const_reverse_rope_view_iterator;

    unencoded_rope_view() noexcept;
    unencoded_rope_view(...) noexcept;
    unencoded_rope_view(..., size_type lo, size_type hi) noexcept;

    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    // etc.

    bool empty() const noexcept;
    size_type size() const noexcept;
```

```cpp
    char operator[](size_type i) const noexcept;

    unencoded_rope_view operator()(size_type lo, size_type hi) const;
    unencoded_rope_view operator()(size_type cut) const;

    template<typename Fn>
    void foreach_segment(Fn && f) const;

    size_type max_size() const noexcept;
    int compare(unencoded_rope_view rhs) const noexcept;

    unencoded_rope_view & operator=(unencoded_rope const & r) noexcept;

    void swap(unencoded_rope_view & rhs) noexcept;
};
```

`operator<<`, and the equality and relational operators are also defined.

`text::unencoded_rope_view` can be constructed from any of the following:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer
- Any range of graphemes built on an underlying range of `char` modeling `ContigGraphemeRange` (e.g. contiguous-storage types from the `text` layer)

All constructors are non-`explicit`, except the generic ones accepting `CharRange` and `ContigGraphemeRange` types.

Assignment is defined for the same types as construction.

Slicing operations are the same as for the previous types, and do not allocate:

```
text::unencoded_rope_view const s("some text");

assert(s(2, 9) == "me text"); // slice
assert(s(2, -1) == "me tex");
assert(s(4) == "some");       // prefix
assert(s(-4) == "text");      // suffix
```

Equality and relational operators are only defined between `text::unencoded_rope_views`. The implicit conversions of many types to `text::unencoded_rope_view` implies comparison to these types:

- `char const *`
- Types from the `string` layer

Comparison to `text` layer types is not allowed.

text::repeated_string_view

```
struct repeated_string_view
{
    using iterator = detail::const_repeated_chars_iterator;
    using const_iterator = detail::const_repeated_chars_iterator;
    using reverse_iterator = detail::const_reverse_repeated_chars_iterator;
    using const_reverse_iterator =
        detail::const_reverse_repeated_chars_iterator;

    constexpr repeated_string_view() noexcept;

    BOOST_TEXT_CXX14_CONSTEXPR
    repeated_string_view(..., size_type count) noexcept;

    constexpr const_iterator begin() const noexcept;
    constexpr const_iterator end() const noexcept;
    // etc.
```

```
    constexpr string_view view() const noexcept;
    constexpr size_type count() const noexcept;

    BOOST_TEXT_CXX14_CONSTEXPR char operator[](size_type i) const noexcept;

    unencoded_rope_view operator()(size_type lo, size_type hi) const;
    unencoded_rope_view operator()(size_type cut) const;

    constexpr bool empty() const noexcept { return view_.empty(); }
    constexpr size_type size() const noexcept;

    BOOST_TEXT_CXX14_CONSTEXPR void
    swap(repeated_string_view & rhs) noexcept;
};
```

`operator<<` and the equality operators are also defined.

`text::repeated_string_view` can be constructed from anything that `text::string_view` can be constructed from (plus a repetition count).

```
std::cout << text::repeated_string_view("    ", indent);
```

```
std::cout << text::repeat("    ", indent);
```

Generalized `string` Layer Operations

- Implicit conversions to view types are the norm
- Explicit conversions to string types are the norm
- Taking `text` layer types and using them with `string` layer types requires more verbosity
- Allocating `operator+` is defined for almost all types, but not between pairs of view types
- Non-allocating slice operations are defined for all types
- Unformatted-output `operator<<` is defined for all types
- All variants of `begin()`, `end()`, `rbegin()`, and `rend()` are defined for all types
- `operator+=`, `insert()`, `erase()`, and `replace()` are defined for the non-view types

These rules form a consistent and highly inter-operable ecosystem of unencoded string types.

Here's how you might choose among the types in the `string` layer.

If I need ...	... my string type is:
to manipulate strings entirely at compile time	`text::string_view`
to capture a reference to a string that will outlive the reference, without allocating	`text::string_view`
a mutable string with efficient mutation at the end of the string	`text::string`

If I need ...	... my string type is:
a mutable string with efficient mutation at any point in the string	`text::unencoded_rope`
a string with contiguous storage	`text::string` or `text::string_view`
a null-terminated string	`text::string`
a mutable string the size of a single pointer	`text::unencoded_rope`

If I need ...	... my string type is:
a thread-safe string	`text::unencoded_rope`
a string with the small-object optimization	`text::string`
a string with copy-on-write semantics	`text::unencoded_rope`

If I need ...	... my string type is:
to capture `char const *s`, `text::string_views`, and `text::strings` in a function parameter	`text::string_view`
to capture `char const *s` and any type in the `string` layer	`text::unencoded_rope_view`

The Unicode layer.

- UTF-8-oriented
- Transcoding iterators that do UTF-8 to/from UTF-16 or UTF-32
- Normalization
- Text segmentation algorithms
- Collation, including tailored collation
- The bidirectional algorithm
- Case mapping (WIP)
- Searching (WIP, probably involves collation folding)

Some Terminology

A code unit is the lowest-level datum-type in your Unicode data. Examples are a `char` in UTF-8 and a `uint32_t` in UTF-32.

A code point is a 32-bit value that represents a single Unicode value. Examples are U+0041/"A"/"LATIN CAPITAL LETTER A" and U+0308/" ̈"/"COMBINING DIAERESIS".

An extended grapheme cluster (or just "grapheme") is a series of one or more code points that looks to an end user like a single glyph or character. For example, The two code units above together look like this: "Ä".

UTF-8

I have yet to find a measurable runtime cost to doing repeated transcoding to/from UTF-8, except when your program does only or mostly transcoding.

UTF-8 is the most compact representation if you look across all languages. CJK code points are an important counterexample.

As part of the "radical simplicity" goal I stated earleir, I'm sticking to UTF-8 only for now. This may have to change one day.

Transcoding

There are four transcoding bidirectional iterators:

- `text::utf8::from_utf32_iterator`
- `text::utf8::to_utf32_iterator`
- `text::utf8::from_utf16_iterator`
- `text::utf8::to_utf16_iterator`

Each has its own `make` function. For instance, `text::utf8::make_to_utf32_iterator()`.

Example usage:

```
std::vector<char> chars = {/* ... */};
for (auto it = text::utf8::make_to_utf32_iterator(chars.begin()),
          end = text::utf8::make_to_utf32_iterator(chars.end());
    it != end; ++it) {
    uint32_t cp = *it;
    // Use cp here...
}
```

Error handling is customizable by providing a template parameter:

```
template<typename Iter, typename ErrorHandler = use_replacement_character>
struct from_utf32_iterator;
```

The transcoding iterators follow the Unicode standard's recommendation on how to handle errors (produce the replacement character), and how often to produce errors from a broken encoding.

Replacement characters are produced at the same places in the sequence whether moving forward or backward.

There are also some utilites that make transcoding more convenient:

- `text::utf8::from_utf32_inserter`
- `text::utf8::from_utf32_back_inserter`
- `text::to_string(CPIter first, CPIter last)`
- `text::utf32_range`

```cpp
std::vector<char> chars = {/* ... */};
for (auto cp : text::utf32_range(chars)) {
    // Use cp here....
}
```

Normalization

The Unicode standard requires that

U+00C4 Ä (LATIN CAPITAL LETTER A WITH DIAERESIS)

must be treated as equivalent to

U+0041 A (LATIN CAPITAL LETTER A) U+0308 ̈ (COMBINING DIAERESIS)

To make it possible to define equality on Unicode strings, the two strings must be normalized, using the same normalization form.

There are four official Unicode normalization forms: NFD, NFC, NFKD, and NFKC.

NFC is the most compact. For this reason, it is the recommended normalization form for the Web, according to W3C.

There is another unofficial normalization form, FCC, that is very similar to NFC.

```
template<typename CPIter, typename Sentinel>
bool normalized_nfd(CPIter first, Sentinel last) noexcept

template<typename CPRange>
bool normalized_nfd(CPRange const & r) noexcept
```

Normalization checks can do a quick-check, which is sometimes indeterminate. In the indeterminate case, the sequence must be normalized and compared to the original input.

```cpp
template<typename CPIter, typename OutIter>
OutIter normalize_to_nfd(CPIter first, Sentinel last, OutIter out);

template<typename CPRange, typename Sentinel, typename OutIter>
OutIter normalize_to_nfd(CPRange const & r, OutIter out);
```

These do not check whether normalization is actually required. They are simply C++ standard library-style algorithms that write the result to an out-iterator.

```
void normalize_to_nfd(string & s);
```

This convenience overload uses the quick-check from `normalized_nfd()` to avoid normalization in those cases in which the quick check works, and uses the capacity of the given string when the normalized form of s fits.

The API for the other normalization forms is identical to this, except for the name of the normalization form.

A partial exception to this is that the `normalized_*()` function for the FCC normalization form is actually spelled `fcd_form()`, because it detects a form that includes more than just FCC.

The Safe-Stream Format

The Unicode standard allows a conforming Unicode implementation to assume that all data are in "Safe-Stream Format". This format limits a single grapheme to at most 32 code points.

The longest grapheme specified in the Unicode data is only 18 code points long. 32 Is intended to be more than enough. Boost.Text adopts this assumption.

Text Segmentation

There are seveal Unicode algorithms that chunk text up in various ways:

- Grapheme clusters
- Words
- Sentences
- Lines
- Paragraphs

The paragraph break algorithm is not an official text segmentation algorithm, but is required in order to implement the bidirectional algorithm.

All the segmentation algorithms operate on sequences of code points, though transcoding iterators can adapt them to sequences of code units.

The word and sentence break algorithms are going to be tailorable eventually, though this is not yet implemented.

```
grapheme_property grapheme_prop(uint32_t cp) noexcept;
```

This returns the grapheme property associated with the given code point.

```cpp
template<typename CPIter, typename Sentinel>
CPIter prev_grapheme_break(CPIter first, CPIter it, Sentinel last) noexcept;

template<typename CPRange, typename CPIter>
auto prev_grapheme_break(CPRange & range, CPIter it) noexcept;
```

Returns the code point at the beginning of the grapheme in which `it` falls, even if `it` is already at the beginning of a grapheme.

```
template<typename CPIter, typename Sentinel>
CPIter next_grapheme_break(CPIter first, Sentinel last) noexcept;

template<typename CPRange>
auto next_grapheme_break(CPRange & range) noexcept;
```

Returns the next grapheme break after `first`. As a precondition, `first` must be at a grapheme break.

```cpp
template<typename CPIter, typename Sentinel>
cp_range<CPIter> grapheme(CPIter first, CPIter it, Sentinel last) noexcept;

template<typename CPRange, typename CPIter>
auto grapheme(CPRange & range, CPIter it) noexcept;
```

Returns bounds of the grapheme in which `it` falls, as a `cp_range` code point range.

```cpp
text::string str = /* ... */;
for (auto cp : text::grapheme(
                   text::utf32_range(str),
                   text::utf8::make_to_utf32_iterator(str.begin() + 8))) {
    // Do something with code point 'cp'...
}
```

```cpp
template<typename CPRange, typename CPIter>
auto graphemes(CPIter first, Sentinel last) noexcept;

template<typename CPRange>
auto graphemes(CPRange & range) noexcept;
```

Returns a lazy range that produces cp_ranges, each of which is a grapheme.

```cpp
text::string str = /* ... */;
for (auto grapheme : text::graphemes(text::utf32_range(str))) {
    for (auto cp : grapheme) {
        // Do something with code point 'cp'...
    }
}
```

Line breaks come in two flavors, "hard" line breaks are required, and non-hard line breaks are essentially line break opportunities. It is up to the application to choose whether a particular non-hard line break is a good place to break.

```cpp
template<typename CPIter, typename Sentinel>
CPIter prev_hard_line_break(CPIter first, CPIter it, Sentinel last) noexcept;

template<typename CPIter, typename Sentinel>
line_break_result<CPIter>
prev_possible_line_break(CPIter first, CPIter it, Sentinel last) noexcept;

template<typename CPIter, typename Sentinel>
auto lines(CPIter first, CPIter last) noexcept;

template<typename CPIter, typename Sentinel>
auto possible_lines(CPIter first, Sentinel last) noexcept;
```

There is one text segmentation iterator, `grapheme_iterator`. It is implementable as an iterator with `O(1)` operations.

There are not other text segmentation iterators, because they do not seem implementable as iterator with `O(1)` operations.

```cpp
template<typename CPIter, typename Sentinel = CPIter>
struct cp_range
{
    cp_range() noexcept {}
    cp_range(CPIter f, Sentinel l) noexcept : first_(f), last_(l) {}

    bool empty() const noexcept { return first_ == last_; }

    CPIter begin() const noexcept { return first_; }
    Sentinel end() const noexcept { return last_; }
};
```

```cpp
template<typename CPIter, typename Sentinel = Iter>
struct grapheme_iterator
{
    using value_type = cp_range<CPIter>;
    using difference_type = std::ptrdiff_t;
    using pointer = value_type const *;
    using reference = value_type;
    using iterator_category = std::bidirectional_iterator_tag;

    grapheme_iterator() noexcept;

    grapheme_iterator(CPIter first, CPIter it, Sentinel last) noexcept;

    reference operator*() const noexcept;
    pointer operator->() const noexcept;

    CPIter base() const noexcept { return grapheme_.begin(); }

    // etc. ...
};
```

```cpp
text::string str = /* ... */;
text::utf32_range range(str);
auto it = text::grapheme_iterator<text::string::iterator>(
    range.begin(), range.begin(), range.end());
auto const end = text::grapheme_iterator<text::string::iterator>(
    range.begin(), range.end(), range.end());

for (; it != end; ++it) {
    // Use grapheme *it here ...
}

// Code point iterators are available via .base().
for (auto cp_it = it.base(), cp_end = end.base();
     cp_it != cp_end; ++cp_it) {
    // Use code point *cp_it here ...
}

// Code units (char) iterators are available via .base().base().
text::string str2(it.base().base(), end.base().base());
assert(str == str2);
```

Collation

Collation is the comparison of two strings for purposes of sorting or searching.

Unicode collation is weird.

There are multiple levels that must be considered when comparing strings. For instance, the primary level is for the most essential difference between two code points. For instance, "a" and "A" are the same at the primary level, but "a" and "b" are different. The former pair are just two ways of writing the same Latin character, whereas the latter pair consists of two distict Latin characters.

Level 2 contains accent differences, level 3 contains variants such as different cases, and level 4 contains punctuation.

To do collation on two strings, you must create a sort key for each, and then do normal lexcographical comparison on the sort keys.

A sort key has the form:

L1-weights[L2-weights[L3-weights[L4-weights]]]

You must use L1 weights, but you can choose to use some or all of the other weight levels (without gaps -- you can't use only L1 and L3).

This means that the length of a sort key for a particular string depends on which levels you want to consider when collating.

The maximum level used to generate your sort keys is known as the collation "strength".

Some examples:

Strength=L1 means "Ignore accents, case, and punctuation"

Strength=L2 means "Ignore case and punctuation"

There are also parameters you can provide to the collation algorithm that create variations such as "Ignore accents, but do consider case".

A peculiarity of Unicode collation is that even though there is a default collation that works for many scripts, languages, and other use cases, it does not work for every use case.

Collation that works for one language often does not work for another.

This means that there needs to be a means available to users of the Unicode collation algorithm of tailoring collation to a particular language or use case.

Boost.Text support the LDML format for specifying collation tailoring. This is what ICU uses.

Examples:

```
[normalization on]
[reorder Grek]
```

```
&N<ñ<<<Ñ
&C<ch<<<Ch<<<CH
&l<ll<<<Ll<<<LL
```

Boost.Text comes with a parser for the LDML tailoring format, and all the tailoring data files that come with ICU, so it can be used with (nearly?) every Unicode language out of the box.

```cpp
template<typename CPIter>
text_sort_key collation_sort_key(
    CPIter first,
    CPIter last,
    collation_table const & table,
    collation_strength strength = collation_strength::tertiary,
    case_first case_1st = case_first::off,
    case_level case_lvl = case_level::off,
    variable_weighting weighting = variable_weighting::non_ignorable,
    l2_weight_order l2_order = l2_weight_order::forward);
```

Another overload exists that takes a code point range.

```
collation_table default_collation_table();

collation_table tailored_collation_table(
    string_view tailoring,
    string_view tailoring_filename = "",
    parser_diagnostic_callback report_errors = parser_diagnostic_callback(),
    parser_diagnostic_callback report_warnings =
        parser_diagnostic_callback());
```

Typical usage:

```
text::collation_table table = text::tailored_collation_table(
    text::data::af::standard_collation_tailoring());
```

The resulting `text::collation_table` object has the semantics of a `std::shared_ptr<T const>`. It is immutable and cheap to copy.

Collation tailoring is quite expensive for some languages, typically the CJK language tailorings, sometimes as much as a multiple seconds.

There is serialization of collation tables to/from a buffer or to/from a `boost::filesystem::path`.

Unicode collation requires the NFD normalization form, one of the least compact normalization forms. However, there is a variant that is defined in Unicode Technical Note #5 that allows one to use an alternate normalization form called FCC.

FCC is very similar to NFC, except that it is less compact in a few cases. Boost.Text's collation implementation relies on the inputs being in the FCC normalization form.

The Bidirectional Algorithm

This algorithm is very complicated. It handles the needed changes in left-to-right or right-to-left direction required for printing text that contains multiple languages.

For, example, assume the uppercase letters here are from a right-to-left language like Arabic or Hebrew:

`"car means CAR."` should be printed as `"car means RAC."`.

In addition, sometime a code point must be replaced by a code point that is its mirror-image, such as `']'` with `'['`.

Also, the bidirectional algorithm uses the line-break algorithm.

```cpp
struct next_hard_line_break_callable
{
    template<typename CPIter>
    CPIter operator()(CPIter first, CPIter last) noexcept
    {
        return next_hard_line_break(first, last);
    }
};
```

```cpp
template<typename CPIter>
struct bidirectional_subrange
{
    using iterator = /* unspecified */;

    bidirectional_subrange() noexcept {}
    bidirectional_subrange(iterator first, iterator last) noexcept :
        first_(first),
        last_(last)
    {}

    bool empty() const noexcept { return first_ == last_; }
    iterator begin() const noexcept { return first_; }
    iterator end() const noexcept { return last_; }
};
```

```
template<
    typename CPIter,
    typename OutIter,
    typename NextLineBreakFunc = next_hard_line_break_callable>
OutIter bidirectional_order(
    CPIter first,
    CPIter last,
    OutIter out,
    NextLineBreakFunc && next_line_break = NextLineBreakFunc{});
```

`bidirectional_order()` produces a sequence of subranges. The iterator type of `bidirectional_subrange` is a variant-like type that may be a forward or reverse iterator.

The replacement of individual an code point is represented by a single-code-point range.

```
text::string str = /* ... */;
text::utf32_range as_utf32(str);
std::vector<text::bidirectional_subrange<char const *>> bidi_segments;
text::bidirectional_order(as_utf32.begin(), as_utf32.end(),
                          std::back_inserter(bidi_segments));
std::vector<uint32_t> bidi_ordered_cps;
for (auto subrange : bidi_segments) {
    std::copy(subrange.begin(), subrange.end(),
              std::back_inserter(bidi_ordered_cps));
}
```

This will get nicer once there's a lazy range version (that is itself range-friendly).

Future Unicode Efforts

Case mapping, like `to_upper()`, `to_lower()`, and `to_title()` are planned, but are not yet implemented.

Collation-based searching is also planned but not yet implemented.

Unicode Database

The data needed to drive all the Unicode algorithms described so far is 2.3MB. This does not include case mapping data (which are small), and can be compressed further.

Unicode Layer Testing

Lots of hand-written tests exist for the `string` and `text` layers, including fuzz testing.

The Unicode website has test files containing numerous test cases for checking conformance of a Unicode implementation.

Boost.Text uses all the test data available for each of its Unicode algorithm implementations. This amounts to more than a million individual checks.

The `text` layer.

- `text::text`
- `text::text_view`
- `text::rope`
- `text::rope_view`

text::text

```
struct text
{
    using iterator =
        grapheme_iterator<utf8::to_utf32_iterator<char *, char *>>;
    using const_iterator = grapheme_iterator<
        utf8::to_utf32_iterator<char const *, char const *>>;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    text();
    text(...);

    text & operator=(...);

    iterator begin() noexcept;
    iterator end() noexcept;
    // etc.

    bool empty() const noexcept;
    int storage_bytes() const noexcept;
    int capacity() const noexcept;
    int distance() const noexcept;
    int max_size() const noexcept;
```

```
    void clear() noexcept;

    iterator insert(iterator at, ...);
    text & erase(...) noexcept;
    text & replace(text_view old_substr, ...);

    void reserve(int new_size);
    void shrink_to_fit();
    void swap(text & rhs) noexcept;

    string extract() && noexcept;
    void replace(string && s) noexcept;

    text & operator+=(...);
};
```

`operator+`, `operator<<`, and the equality operators are also defined.

`text::text` can be constructed from any of the following:

- `char const *`
- Any range of `char` modeling `CharRange` (e.g. `std::string`)
- Types from the `string` layer
- Types from the `text` layer

All constructors are `explicit`, except the one taking `char const *`.

Assignment is defined for the same types as construction.

`text::text` has no slicing operations.

`insert()` is defined for `iterator` positions only (no integral indices), since random access is not provided for `text::text`.

The `insert()` overloads accept the same types that the constructors and assignment operators take.

Same with with `replace()` and `operator+=`.

Equality operators are only defined between `text::text` and the other `text` layer types.

Comparison to `string` layer types is not allowed with a simplified syntax, because `string` layer types know nothing about encoding, normalization, or collation.

The iterator types are bidirectional, not random access.

There is no `size()` data member, because it would have to be `O(N)`, due to the bidirectional iterators.

```
int storage_bytes() const noexcept;
```

Instead, there is a member that gives the total size of storage in bytes.

```
int distance() const noexcept;
```

There is also a an `O(N)` member that gives the total number of elements.

```
string extract() && noexcept;
void replace(string && s) noexcept;
```

There are two members that allow you to steal the guts, a `text::string`, and then replace them.

The underlying text is assumed to be UTF-8 encoded. This is a safe assumption, because the transcoding iterator used internally automatically inserts the Unicode replacement character.

The underlying text is kept normalized, because this allows `operator==` to have an efficient implementation.

The normalization form is always FCC, since this is nearly optimal in its use of space, and using this form means that normalization can be skipped during collation.

Normalized strings are not closed under most string operations, including insertion, erasure, and concatenation, so portions of a `text::text` must be re-normalized during these operations.

text::text_view

```
struct text_view
{
    using iterator = grapheme_iterator<
        utf8::to_utf32_iterator<char const *, char const *>>;
    using const_iterator = iterator;
    using reverse_iterator = std::reverse_iterator<const_iterator>;
    using const_reverse_iterator = reverse_iterator;

    text_view() noexcept;
    text_view(...) noexcept;

    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    // etc.

    bool empty() const noexcept;
    int storage_bytes() const noexcept;
    int distance() const noexcept;
    int max_size() const noexcept;
    void swap(text_view & rhs) noexcept;
};
```

`operator<<`, and the equality operators are also defined.

`text::text_view` can be constructed only from a `text::text`, `text::text_view`, or a pair of `text::text::iterator`s.

All constructors are non-`explicit`.

Assignment is defined for the same types as construction.

text::rope

```
struct rope
{
    using iterator = grapheme_iterator<utf8::to_utf32_iterator<
        detail::const_rope_iterator,
        detail::const_rope_iterator>>;
    using const_iterator = iterator;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = reverse_iterator;

    rope();
    rope(...);

    rope & operator=(...);

    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    // etc.

    bool empty() const noexcept;
    size_type storage_bytes() const noexcept;
    size_type distance() const noexcept;
    size_type max_size() const noexcept;
```

```cpp
    template<typename Fn>
    void foreach_segment(Fn && f) const;

    bool equal_root(rope rhs) const noexcept;

    void clear() noexcept;

    iterator insert(iterator at, ...);
    rope & erase(rope_view rv);
    rope & replace(rope_view old_substr, ...);

    void swap(rope & rhs) noexcept;

    unencoded_rope extract() && noexcept;
    void replace(unencoded_rope && ur) noexcept;

    rope & operator+=(...);
};
```

`operator+`, `operator<<`, and the equality operators are also defined.

`text::rope` can be constructed from all the same types as `text::text` (just about anything string- or text-like).

All the `text::rope` operations are defined for the same operand types as the `text::text` operations.

text::rope_view

```
struct rope_view
{
    using iterator = grapheme_iterator<utf8::to_utf32_iterator<
        detail::const_rope_view_iterator,
        detail::const_rope_view_iterator>>;
    using const_iterator = iterator;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = reverse_iterator;

    using const_rope_iterator = grapheme_iterator<utf8::to_utf32_iterator<
        detail::const_rope_iterator,
        detail::const_rope_iterator>>;

    rope_view() noexcept;
    rope_view(...) noexcept;
    rope_view(const_rope_iterator first, const_rope_iterator last) noexcept;

    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    // etc.
```

```
    bool empty() const noexcept;
    size_type storage_bytes() const noexcept;
    size_type distance() const noexcept;
    size_type max_size() const noexcept;

    template<typename Fn>
    void foreach_segment(Fn && f) const;

    void swap(rope_view & rhs) noexcept;
};
```

`operator<<`, and the equality operators are also defined.

`text::rope_view` can be constructed only from a `text::rope`, `text::rope_view`, or a pair of `text::rope::iterators`.

All constructors are non-`explicit`.

Assignment is defined for the same types as construction.

All the slides about how to pick the right `string` layer type apply to how to pick the right `text` layer type. The `text` layer mostly just adds encoding and normalization guarantees.

The `text` layer adheres to the radical simplicity goal of Boost.Text. There are not multiple choices for encoding, normalization, or allocation.

We'll see if, and to what extent, this survives Boost review and user experience, but I think this is an important goal.

Some Other Stuff

There are also a few implementation details that rise to the level of general usefulness.

- `segmented_vector<T>`
- `trie<Key, Value>`

segmented_vector<T>

```
template<typename T>
struct segmented_vector
{
    using iterator = detail::const_vector_iterator<T>;
    using const_iterator = detail::const_vector_iterator<T>;
    using reverse_iterator = detail::const_reverse_vector_iterator<T>;
    using const_reverse_iterator = detail::const_reverse_vector_iterator<T>;

    segmented_vector() noexcept;

    template<typename Iter>
    segmented_vector(Iter first, Iter last);
    segmented_vector(std::initializer_list<T> il);

    segmented_vector & operator=(std::initializer_list<T> il);

    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    // etc.

    bool empty() const noexcept;
    size_type size() const noexcept;
    size_type max_size() const noexcept;
```

```
T const & operator[](size_type n) const noexcept;

template<typename Fn>
void foreach_segment(Fn && f) const;

int compare(segmented_vector rhs) const noexcept;
bool operator==(segmented_vector rhs) const noexcept;
bool operator!=(segmented_vector rhs) const noexcept;
bool operator<(segmented_vector rhs) const noexcept;
bool operator<=(segmented_vector rhs) const noexcept;
bool operator>(segmented_vector rhs) const noexcept;
bool operator>=(segmented_vector rhs) const noexcept;

bool equal_root(segmented_vector rhs) const noexcept;
void clear();

segmented_vector & push_back(T t);
```

```
    const_iterator insert(const_iterator at, T t);
    const_iterator insert(const_iterator at, std::vector<T> t);
    template<typename Iter>
    const_iterator insert(const_iterator at, Iter first, Iter last);

    const_iterator erase(const_iterator at);
    const_iterator erase(const_iterator first, const_iterator last);
    segmented_vector & replace(const_iterator at, T t);
    segmented_vector &
    replace(const_iterator first, const_iterator last, std::vector<T> t);
    template<typename Iter>
    segmented_vector & replace(
        const_iterator old_first,
        const_iterator old_last,
        Iter new_first,
        Iter new_last);

    void swap(segmented_vector & rhs);
};
```

This API is very similar to that of `text::unencoded_rope`, except that is supports more types than `char` for the element-type.

The `replace()` API was retained, since it is considerably more efficient to do a replace operation than to do an erasure followed by an insertion.

A few interface changes were made to make the type more like `std::vector`.

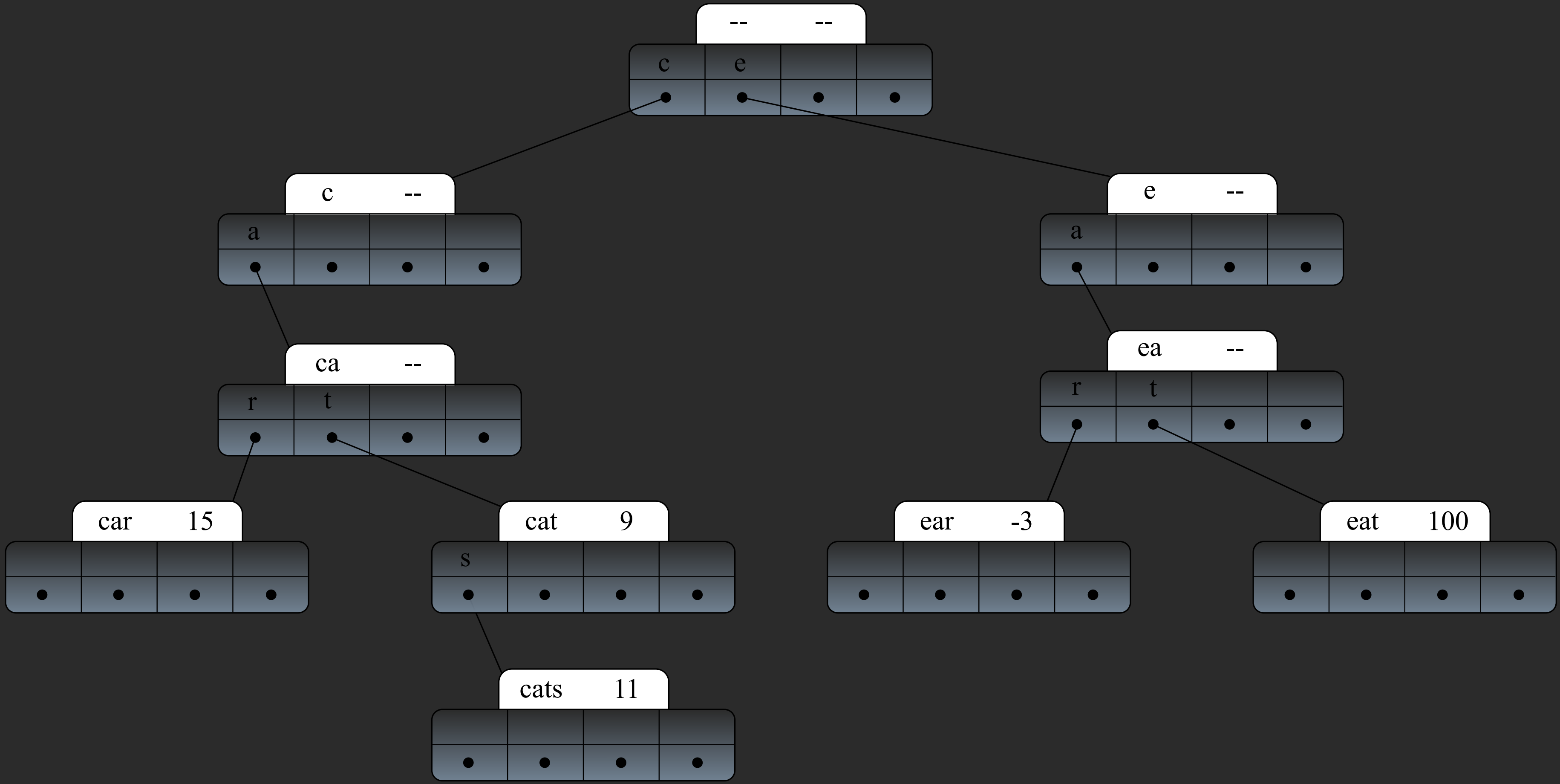
Again, this make writing undo systems trivial:

```
std::vector<text::segmented_vector<T>>::iterator
new_undo_state(std::vector<text::segmented_vector<T>> & history,
               std::vector<text::segmented_vector<T>>::iterator it)
{
    history.erase(std::next(it), history.end());
    return history.insert(history.end(), history.back());
}

std::vector<text::segmented_vector<T>>::iterator
undo(std::vector<text::segmented_vector<T>> const & history,
     std::vector<text::segmented_vector<T>>::iterator it)
{
    if (it != history.begin())
        --it;
    return it;
}

std::vector<text::segmented_vector<T>>::iterator
redo(std::vector<text::segmented_vector<T>> const & history,
     std::vector<text::segmented_vector<T>>::iterator it)
{
    if (std::next(it) != history.end())
        ++it;
    return it;
```

```cpp
template<typename T, typename Func>
std::vector<text::segmented_vector<T>>::iterator
mutate_vec(std::vector<text::segmented_vector<T>> const & history,
           std::vector<text::segmented_vector<T>>::iterator it,
           Func && f)
{
    it = new_undo_state(history, it);
    std::forward<Func>(f)(*it);
    return it;
}
```

`trie`

A trie is a prefix-tree that associates keys (which must be sequences) with values. Each node represents a prefix, and each child node represents a possible next element of the key.



{("car", 15), ("cat", 9), ("cats", 11), ("ear", -3), ("eat", 100)}.

trie<Key, Value>

```
template<typename Key, typename Value, typename Compare = less>
struct trie
{
    using value_type = Value;
    using key_compare = Compare;
    using key_element_type = typename Key::value_type;

    trie();
    trie(Compare const & comp);

    template<typename Iter>
    trie(Iter first, Iter last, Compare const & comp = Compare());
    template<typename Range>
    explicit trie(Range r, Compare const & comp = Compare());
    trie(std::initializer_list<value_type> il);

    trie & operator=(std::initializer_list<value_type> il);
    bool empty() const noexcept;
    size_type size() const noexcept;
    template<typename KeyRange>
    bool contains(KeyRange const & key) const noexcept;
```

```
template<typename KeyIter>
match_result longest_subsequence(KeyIter first, KeyIter last) const
    noexcept;
template<typename KeyRange>
match_result longest_subsequence(KeyRange const & key) const noexcept;
template<typename KeyIter>
match_result longest_match(KeyIter first, KeyIter last) const noexcept;
template<typename KeyRange>
match_result longest_match(KeyRange const & key) const noexcept;
```

```
template<typename KeyElementT>
match_result extend_subsequence(match_result prev, KeyElementT e) const
    noexcept;
template<typename KeyIter>
match_result
extend_subsequence(match_result prev, KeyIter first, KeyIter last) const
    noexcept;

template<typename OutIter>
OutIter copy_next_key_elements(match_result prev, OutIter out) const;

template<typename KeyRange>
optional_ref<value_type const> operator[](KeyRange const & key) const
    noexcept;
```

```
void clear() noexcept;

template<typename KeyRange>
optional_ref<value_type> operator[](KeyRange const & key) noexcept;

template<typename KeyIter>
bool insert(KeyIter first, KeyIter last, Value value);
template<typename KeyRange>
bool insert(KeyRange const & key, Value value);
template<typename Char, std::size_t N>
bool insert(Char const (&chars)[N], Value value);
template<typename Iter>
void insert(Iter first, Iter last);
template<typename Range>
bool insert(Range const & r);
void insert(std::initializer_list<value_type> il);
```

```
    template<typename KeyRange>
    bool erase(KeyRange const & key);

    void swap(trie & other);
};
```

Lookups are comparable to hashing containers, except that a trie additionally allows one to easily do longest-match and match-extension queries.

`operator[]` works differently from how it works in `std::map`-like containers.

```
trie<text::string, int> t = /* ... */;
auto element = t["foo"];
if (element) { // if I'm not sure, or if 't' is const
    int value = element;
}
```

```
trie<text::string, int> t = /* ... */;
int value = t["foo"]; // if I'm sure "foo" is in 't'
```

`trie` is not a container. It has no iterators.

`trie_map<Key, Value>` and `trie_set<Key>`

These are very similar to `trie`, except that they have iterators, and `trie_set` has a more set-like interface.

The need to support iterators has a runtime cost, so without a need for iterators, `trie` should be preferred.

Miscellaneous Algorithms

```
template<typename BidiIter, typename Sentinel, typename T>
BidiIter find_not(BidiIter first, Sentinel last, T const & x);
```

This just seems to me to be a logical addition to the existing algorithms `find()`, `find_if()`, and `find_if_not()`.

Consider this case: you want to find a value at or before a current iterator `it`:

```
// assume some sequence seq, and some value x
auto const rfirst = std::make_reverse_iterator(std::next(it));
auto const rlast = std::make_reverse_iterator(seq.begin());
auto rit = std::find(rfirst, rlast, x);
if (rit != rlast)
    it = (--rit).base();
```

That's not the greatest code. You only have to increment *one* of the bounds of `(seq.begin(), it]` for `find()` to search that interval in reverse. You also have to decrement the result before getting its `base()`, or you'll be pointing to the wrong result ... but you cannot do this without checking that `rit` does not point to `rlast`.

We can do better:

```
template<typename BidiIter, typename T>
BidiIter find_backward(BidiIter first, BidiIter last, T const & x);
template<typename BidiIter, typename T>
BidiIter find_not_backward(BidiIter first, BidiIter last, T const & x);
template<typename BidiIter, typename Pred>
BidiIter find_if_backward(BidiIter first, BidiIter last, Pred p);
template<typename BidiIter, typename Pred>
BidiIter find_if_not_backward(BidiIter first, BidiIter last, Pred p);
```

```
it = find_backward(seq.begin(), std::next(it));
```

If we only want to look strictly before `it`, we can do that too:

```
it = find_backward(seq.begin(), it);
```

Even though we still have to increment only one bound of `(seq.begin(), it]` in one of the cases, now that the code is so much terser, that difference is easy to notice. In the more verbose code, that difference would get lost in the noise.

I find this pattern of code to be relatively common. I want to find all the runs of values that are the same value, or that match some predicate:

```
// assume some sequence seq with value type value_type, and some value x
auto it = seq.begin();
while (it != seq.end()) {
    it = std::find(it, seq.end(), x);
    auto next = std::find_if_not(
        it, seq.end(),
        [](value_type const & value) { return value == x; });
    // Use [it, next) ...
    it = next;
}
```

```cpp
template<typename Iter, typename Sentinel = Iter>
struct foreach_subrange_range
{
    using iterator = Iter;
    using sentinel = Sentinel;

    foreach_subrange_range();
    foreach_subrange_range(iterator first, sentinel last);

    iterator begin() const noexcept;
    sentinel end() const noexcept;
};

template<typename FwdIter, typename Sentinel, typename T, typename Func>
void foreach_subrange(FwdIter first, Sentinel last, T const & x, Func f);

template<typename FwdIter, typename Sentinel, typename Pred, typename Func>
void foreach_subrange_if(FwdIter first, Sentinel last, Pred p, Func f);
```

Questions?

https://github.com/tzlaine/text