Generic Conversion and Segmentation for Ranges

A Solution for Unicode

Mathias Gaunard

MetaScale Inc.

Boostcon 2011

Context

- Google Summer of Code 2009
- Available on the Boost Sandbox in SOC/2009
- Doc at `http://mathias.gaunard.com/unicode/doc/html`

Should it be submitted for review? – Feedback welcome

What's Unicode?

- A character set that unifies all character sets for all languages
  More than 1 million "entries", or **code points** (21-bit mapping)
- A set of data attached to each code point
  - category info
  - possible decompositions
  - uppercase/lowercase/case-folded version
- A mechanism to combine code points to create **combining character sequences**, and the associated algorithms to deal with them
- Algorithms to delimit graphemes, words, sentences, possible line breaks
- Collation algorithms for advanced comparison and sorting

UTF encodings

21-bit code points impractical, various encodings available:

- UTF-8, encode a code point as a variable-sized sequence of 1 to 4 8-bit code units – also has lots of nice properties
- UTF-16, encode a code point as one or two 16-bit code units
- UTF-32, enode a code point as a single 32-bit code unit

Variable-width radically different from legacy character sets.

Combining character sequences

- Any number of combining code points can be appended after a non-combining code point
  Forms a single combining character sequence
- No hard limit on the number of combinations, some exist as precomposed code points, some do not (e.g. some Navajo characters)
- The same combining character sequence can have lots of different representations

- Canonical ordering of combining code points
- Normalization: fully decomposed, fully decomposed (NFD) then recomposed (NFC)

Example

The ệ character can be represented in several ways:

U+1ec7	e with circumflex and dot below
U+1eb9 U+0302	e with dot below + combining circumflex
U+0065 U+0323 U+0302	e + combining dot below + combining circumflex

U+0065 U+0302 U+0323 also canonically equivalent, but not canonically ordered.

What's a character?

Different approximations:

- Code unit
- Code point
- Combining character sequence
- Grapheme cluster

Grapheme cluster *real* character for the end-user, but not necessarily for the programmer.
Some combining character sequences aren't graphemes and some graphemes aren't combining character sequences – different notions.

How does Unicode translate into a library?

A table of properties for each code point:

- Fairly large data, needs to remove redundancy
- May need to be tailored for particular locales
- A dynamically-linked library with a stable and simple ABI makes sense
- Database needs to be at least backward-compatible

Low-level interface, not what we want to provide to the user, but still needs to be there.

Unicode Character Database

Boost.Unicode generates one:

- Spirit Classic parser
- Two-level memory structure, but redundancy removal not done on a per-property basis
- A function per property that returns that property value for a code point
- Backward compatible but also forward compatible by using functions on the caller side that checks whether the property value is in a known range
- Composition needs prefix (and possibly suffix) trees, needs to come up with a better ABI to expose these

What we need

- Conversion/transformation
  - UTF decoding/encoding
  - Normalization, decomposition, composition
  - Case folding
- Concatenation of normalized strings (normalization not stable by concatenation)
- Segmentation, for any UTF-X range:
  - Code points
  - Combining character sequences
  - Graphemes, words, etc.
- Find closest boundaries from a random position (related to segmentation)
- Substring search and match

Operations on text

Yet another huge monolithic inflexible string type with member functions that do everything, including what you may not want or need?

No, we want:

- To be able to work with any string type, wherever/however it is stored
- To control what memory is allocated, when and how – even be able to avoid allocating freestore memory entirely if possible
- To be able to combine transformations easily and efficiently
- For conversions to be *fast*
- It to be as easy to use and unintrusive as possible

The Solution

- Works with any range
- A conversion **only needs to be written once** and with a **very simple interface** to be used in different ways
- Conversions can be combined, applied eagerly or lazily
- Can exploit parallelism

The Range concept

A range is a type from which you can extract a begin and a past-the-end iterator

- Concept is refined just like the Iterator concepts
- Containers are ranges, `std::pair<It, It>` too
- Terser syntax than iterators, can be used with `BOOST_FOREACH` or C++0x range for-loop
- Boost.Range provides the basis of range primitives, as well as pretty cool range adaptors

Boost.Range adaptors

```cpp
std::vector<int> v = {1, 2, 3, 4};
transformed_range<
    F,
    filtered_range<
        P,
        iterator_range< std::vector<int>::iterator >
    >
> adapted = r | filtered(p) | transformed(f);
```

Returns a range `adapted` that will, as you iterate it, iterate through the elements of the vector `v`, discard elements that do not satisfy the predicate `p` and apply the function `f` on each element.

Unicode adaptors

Like with DSELs, return type is complex, avoid writing it out.

- Use auto
- Don't name the variable

We're going to try to define a mechanism that allows things like encoding conversion or even more complicated operations to be expressed lazily in a similar way.

Converter concept

```
struct Converter
{
    typedef unspecified input_type; // archetype for concept-checking
    typedef unspecified output_type;
    typedef mpl_integral_constant max_output; // optional
    typedef mpl_integral_constant output_alignment; // optional

    template<typename In, typename Out>
    Out ltr(In& begin, In const& end, Out const& out);

    template<typename In, typename Out>
    Out rtl(In const& begin, In& end, Out const& out);
};
```

- Defines a **step** of a conversion that advances `begin` or `end` depending on whether you iterate left-to-right or right-to-left respectively.
- “Consumes” some elements from the `In` range and writes some new ones to `Out`, writing up to `max_output` elements in a single step.

How to use a converter

- Eager evaluation:

```
std::string utf8_data = "Hello World";

std::basic_string<char32> utf32_data;
convert(utf8_data, u8_decoder(), std::back_inserter(utf32_data));

BOOST_FOREACH(char32 cp, utf32_data)
    std::cout << "Code point " << cp << "\n";
```

- Lazy evaluation:

```
std::string utf8_data = "Hello World";

BOOST_FOREACH(char32 cp, adaptors::convert(string, u8_decoder()))
    std::cout << "Code point " << cp << "\n";
```

Two-pass eager evaluation

Don't have to use push_back, can compute the size that you need, allocate the buffer, and convert it there.

```
counting_iterator<size_t> it = convert(
    utf8_data,
    u8_decoder(),
    counting_iterator<size_t>(0)
);

std::vector<char32> utf32_data(it.base());
convert(utf8_data, u8_decoder(), utf32_data.begin());
```

Segmenter concept

```
struct Segmenter
{
    typedef unspecified input_type; // archetype for concept-checking
    typedef unspecified tag_type; // optional

    template<typename In>
    tag_type ltr(In& begin, In const& end);

    template<typename In>
    tag_type rtl(In const& begin, In& end);
};
```

Like a `Converter`, but no output and potentially a tag.

How to use a segmenter

```
std::string utf8_data = "Hello World";

typedef iterator_range<std::string::iterator> sub_range;
BOOST_FOREACH(sub_range cp, segment(utf8_data, u8_segmenter()))
{
    std::cout << "Code point ";
    BOOST_FOREACH(char c, cp)
        std::cout << (int)c << ", ";
    std::cout << "\n";
}
```

BoundaryChecker concept

```
struct BoundaryChecker
{
    typedef unspecified input_type; // archetype for concept-checking

    template<typename In>
    bool operator()(In const& begin, In const& end, In const& pos);
}
```

Returns whether the position `pos` within the range [`begin`, `end`[lies on a particular boundary.

Building boundary checkers and segmenters

- `multi_boundary`: builds a `BoundaryChecker` that tests for a boundary, applies a converter, then checks for another boundary.
- `converter_segmenter`: builds a `Segmenter` from a `Converter` by discarding its input
- `boundary_segmenter`: builds a `Segmenter` from a `BoundaryChecker` by advancing until the boundary
- `converted_segmenter`: builds a `Segmenter` that applies a converter before applying a segmenter – converter must have a max output of 1.

Building converters

- `multi_converter`: builds a `Converter` that applies a converter after another, step output of first must combine well with expected input of second converter
- `converted_converter`: builds a `Converter` that applies a converter after another – first converter must have a max output of 1.
- `codecvt_in_converter` and `codecvt_out_converter`: builds a `Converter` from one direction of a codecvt facet

Converters and segmenters usage

- Can generate a codecvt facet from two converters and two boundary checkers for transparent conversion on std::fstream – slow, don't use it unless you love standard iostreams
- Can build an iterator adaptor that applies the converter step by step – lazy, removes buffering and memory allocation problems
- Can evaluate them in a tight loop – fastest

Unicode converters and segmenters

Basic primitives:

- `cast_converter`
- `u8_decoder`, `u8_encoder`, `u8_boundary`, equivalent `u16_*` ones
- `locale_utf_transcoder`, `utf_locale_transcoder` – built from codecvt facets
- `combining_boundary`
- `grapheme_boundary`
- `decomposer` and `composer`

Transcoding

- Convenience UTF transcoding converters:
  - `utf_decoder` – calls the correct one depending on the size of the elements
  - `utf_encoder<T>` – calls the correct one depending on the size of T
  - `utf_transcoder<T>` – calls `utf_decoder` then `utf_encoder<T>`
- Conversion with other character sets:
  - `latin1_encoder`
  - `locale_decoder`, `locale_encoder`
- Normalization
  - `normalizer`

UTF variants

- `u8_segment` and `u16_segment` – UTF segmenters, two possible implementations
- `u8_combining_boundary`, `u8_grapheme_boundary`, etc.
- `u8_combining_segment`, `u8_grapheme_segment`, etc.
- `u8_normalizer` etc.

String search and match

Two solutions:

- Adapt the range as a range of what you want to match on and pass that to a generic search algorithm
- Tries to match the range as-is but discard matches that do not lie on the expected boundaries

Second solution is typically faster, but needs some wrappers for the search algorithms.

String search example

```cpp
std::string input = "foo\xcc\x82foo";
std::string search = "foo";

// Adapted ranges
auto match
  = algorithm::find_first( adaptors::u8_grapheme_segment(input),
                           adaptors::u8_grapheme_segment(search)
                         );
```

String search example with boundary check

Boost.Unicode provides adapters for models of the Boost.StringAlgo `Finder` concept in order to filter matches that do not satisfy a particular `BoundaryChecker`.

```
// Boundary check
auto finder = make_boundary_finder( algorithm::first_finder(search),
                                    u8_grapheme_boundary()
                                  );
iterator_range<std::string::iterator> match
  = algorithm::find(input, finder);
```

UTF-8 decoding

```
unsigned char b0 = *(begin++);
if((b0 & 0x80) == 0)
    return char32(b0);

unsigned char b1 = *(begin++);
if((b0 & 0xe0) == 0xc0)
    return (char32(b1) & 0x3f) | ((char32(b0) & 0x1f) << 6);

unsigned char b2 = *(begin++);
if((b0 & 0xf0) == 0xe0)
    return (char32(b2) & 0x3f) | ((char32(b1) & 0x3f) << 6)
          | ((char32(b0) & 0x0f) << 12);

unsigned char b3 = *(begin++);
if((b0 & 0xf8) == 0xf0)
    return (char32(b3) & 0x3f) | ((char32(b2) & 0x3f) << 6)
          | ((char32(b1) & 0x3f) << 12) | ((char32(b0) & 0x07) << 18);
```

Vectorized UTF-8 decoding

Is UTF-8 decoding vectorizable?

- Promotion from uint8 to uint32
- Branching
- Data to consume per step is variable-width and interleaved

Not easy to vectorize, and won't necessarily be fast.

Could have a `u8_fast_decoder` that outputs 4 code points aligned on a 16 boundary in one step; or less if not enough data.

UTF-8 decoding with SIMD – Teaser

```
return select( is_eqz(b0 & 0x80),
               b0,
               select( eq(b0 & 0xe0, 0xc0),
                       (b1 & 0x3f) | (b0 & 0x1f) << 6,
                      select( eq(b0 & 0xf0, 0xe0),
                              (b2 & 0x3f) | ((b1 & 0x3f) << 6)
                             | ((b0 & 0x0f) << 12),
                              (b3 & 0x3f) | ((b2 & 0x3f) << 6)
                             | ((b1 & 0x3f) << 12) | ((b0 & 0x07) << 18)
                             )
                      )
              )
;
```