mirek wrote on Mon, 29 May 2017 20:40
I really feel like I am missing something here, but what does it mean "not indexable"?

Well the perceived problem with Utf8 is that you have a string s[i] and s[i+1] are code units, not a code points. It is not indexable.

So the proposed solution is to use Utf32, where s[i] and s[i+1] are code points. Code points are indexable.

But, here is the issue. Most, but not all algorithms fall into opaque ones or glyph based. In opaque ones you care more about the data in bulk and don't really interpret each code unit. In glyph based ones, s[i] and s[i+1] are code points, but that doesn't help you. You need to determine glyph[i] and glyph[i + 1], so you are back to Utf8 levels of boundary determination even with Utf32. You need "indexable" glyphs, not indexable code units, but they are not.

So the solution is to use a string walker.
Quote:
What is the result of 'Get'?

Get return the code unit.

The code still remains largely the same, you go from something like:

```
char* s = string_8bits;
while (s ...) {
    if (*s)
        ....
    s++;
}
```

to

```
StringWalker s = string_utf8;
while (s ...) {
    if (*s)
        ....
    s++;
}
```

In the constructor, you set up the first code unit.

*s, or Get, returns it.

++ seeks the next code unit.

In the same class, or a different one, you implement the GlyphWalker mechanics.

With a GlyphWalker you can do things like "visual" strlen, where you pass it in A C1 C2 B C3 C4, where Cx are punctuation marks, and the first glyph boundary is A and the second B, with the returned length of 2. The same class handles right to left languages, with the help of block boundary getters.

As you may know, I am implementing my own Core-like library, and one of its goals is to have perfect Unicode support. I can help you with some pieces of code, but I need to know what you want to do.

To expand upon the example I have given before, one needs ToUpper.

ToUpper is hard to get small and fast, since there is no rhyme and reason to it. I tested the data and weird values pop up all the time.

What i did is encode all Unicode characters in two tables.

One is a table of words. The entire Unicode spectrum is divided into chunks of 8 to 32 characters. The word table index is the chunk index. If all the characters in the chunk are well behaved, all you need is the original table.

As an example, characters 0 tot 7 have the same properties and no cases, so the upper bits of the word table are zero and the lower bits are the properties.

If the chunk is not well behaved, the upper bits are the index to a second table of qwords and the lower bits are zero.

I tested all chunk sizes and surprisingly 8 characters/chunk occupy the least amount of RAM.

Using the two tables, you get something similar to this ancient code that is no longer up to date:

```
dword _ToUpper(dword c) {
 dword plane = c >> 16;
 if (plane < PLANES) {
  dword group = c / BS;
  dword offs = c % BS;
  word gdata = db1[group];
  if (gdata & 0x8000) {
   gdata &= 0x7FFF;
   return db2[gdata * BS + offs] & 0x3FFFF;
  }
```

```
  else
   return c;
 }
 return c;
}
```

This is the fastest and most compact scheme I could come up with without some RLE or other compression method. But this is meant to serve 1114112 code units across all 17 planes. Not just 2048 characters. Even if you somehow manage to squeeze 1114112 characters into a word table, which is BTW impossible, when in some cases upper/title/lower case are all over 16 bits in size, that is still 1114112 * bytes. My scheme compresses the range down to 131456 bytes across the two tables, with the caveat that for planes that are not in use, planes above 2, c = ToUpper(c).

Did a quick test, and if somehow in the future, all 1114112 are assigned random values, you probably will have a 600+ KiB table.

Maybe you can come up with and even faster or compact scheme, but I wanted it to be near constant cost.
