Subject: Choosing the best way to go full UNICODE
Posted by mirek on Sat, 27 May 2017 14:58:08 GMT
View Forum Message <> Reply to Message

Hi,

last week, for the first time, I have encountered a problem caused by presence of characters outside of UCS2 range (did you know that emoji characters have assigned codepoints? I did not either. Yet our www backend was failing because somebody send "thumbs up" in the message... :)

So perhaps it is time for full support. The mission target is that these characters are correctly loaded/stored to/from LineEdit, DocEdit, EditString, RichText.

Now I can see 3 paths to achieve this:

(1) Extend WString to 32-bits (so that sizeof(wchar) is 4 bytes). This is the most straightforward and right now my favorite. Downsides is that I am a little bit afraid about backward compatibility and that the performance will be worse than WString. Performance is less of an issue because I believe that in most cases, texts are stored as UTF-8 String anyway, but worth checking.

(2) Add 32-bit LString (or "UString"?) and perhaps deprecate WString. Fixes the problem with compatibility, but fattens the API.

(3) Keep WString and somehow fix things to work with surrogate pairs.

Any thoughts?

Mirek

Subject: Re: Choosing the best way to go full UNICODE
Posted by Zbych on Sat, 27 May 2017 18:02:23 GMT
View Forum Message <> Reply to Message

Correct me if I am wrong, but you need to keep wchar 2-bytes long to be compatible with WinApi, ToSystemCharsetW and FromSystemCharsetW depend on this size.
So I would vote for DString (like DWORD).

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Sat, 27 May 2017 18:48:17 GMT
View Forum Message <> Reply to Message

Good point, but I do not think this is a showstopper. Clearly

String FromSystemCharsetW(const wchar *src)

would have to be

String FromSystemCharsetW(const WCHAR *src)

and a bit of fixing would to be done. The advantage is that it would be all compile-time errors.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Sat, 27 May 2017 18:51:13 GMT
View Forum Message <> Reply to Message

A good contraargument is search in *.h files for WString. There is quite a lot of functions that would need DString variant implemented.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 29 May 2017 12:36:09 GMT
View Forum Message <> Reply to Message

Hi Mirek!

Welcome to my problems, circa 1-2 years in of using U++ :lol:.

I didn't know if you remember, but I was pressuring you left and right to fix these issues, with occasional fixes and what not.

Some fixes managed to get in, but there was no real interest in it, so my solution was to eventually deprecate whole String support in U++, using it as a pure storage. String + custom conversions methods + some custom things, like ToUpper, fixes the problems.

U++ is at least a decade behind on Unicode, with absolutely no good reason. It is not like the issue can't all be fixed in like 3 weeks. I'm sure it would fail 100% of the more difficult compatibility tests.

The good news is that these issues are so rare that you are extraordinarily unlikely to encounter them as an American or European.

As anecdotal proof, this is the first time you encountered issues.

I'm using a very dense data representation method, but even like this, full support is almost 200 KiB of data in every single executable. This is without more advanced stuff, like scripts. I believe 180 KiB is the minimum data that can be stored for the first 3 Unicode planes, this vs. your 2048 entry table in U++, but this gives you lower, upper, title case and Unicode category. I have unit-tests covering these, making sure they are always correct.

String and WString are used for storage. Mostly String.

Contrary to popular belief, Unicode code points are not indexable, so DString does not help. Unicode is glyph based. This + DString being the least beneficent method of storage makes DString not really needed.

In most cases, you can treat the String as opaque.

When not, the ideal solution is an amortized string walker and light algorithm rewrites. You can create an indexed string walker, that seeks, but if you ask it for indices in order, the seek overhead is basically zero. This is only needed for glyph based algorithms, like case conversion and rendering. Or a traditional string walker without indices, a bit faster, but a lit uglier to use, with Get and ++/-- methods.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 29 May 2017 17:40:03 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 29 May 2017 14:36
Contrary to popular belief, Unicode code points are not indexable, so DString does not help. Unicode is glyph based. This + DString being the least beneficent method of storage makes DString not really needed.

I really feel like I am missing something here, but what does it mean "not indexable"?

Quote:
When not, the ideal solution is an amortized string walker and light algorithm rewrites. You can create an indexed string walker, that seeks, but if you ask it for indices in order, the seek overhead is basically zero. This is only needed for glyph based algorithms, like case conversion and rendering. Or a traditional string walker without indices, a bit faster, but a lit uglier to use, with Get and ++/-- methods.

What is the result of 'Get'?

Mirek

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 30 May 2017 08:31:03 GMT
View Forum Message <> Reply to Message

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.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Tue, 30 May 2017 09:03:01 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 30 May 2017 10:31
Get return the code unit.

What is 'code unit' in your teminology?

If I go by definition here: https://en.wikipedia.org/wiki/Character_encoding, it does not make sense with your explanation.

Now to explain what I am trying to do now is to provide some means to add full support to U++ editors in future. To that end, I need to define "cursor position" within the UNICODE text.

So far I believe that a "index of UNICODE codepoint" is a good cursor position. Alternatively, grapheme could be a good cursor position too, but then sometimes you would like to edit grapheme internals, like remove some combining mark, which could be difficult if grapheme is indivisible.

Anyway, it is definitely somthing to think about.

Mirek

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 30 May 2017 09:23:15 GMT
View Forum Message <> Reply to Message

I meant code point.

Get is the code point.

And what I claimed is that having code point = code unit only half-way solves the problem and you get the least favorite and advantageous encoding in DString/Utf32.

The index of Utf code-point can be easily solved with a string walker that can work for any encoding, from utf8 to 32 and there is no need to tie yourself down with DString supremacy.

Because once you reach "full" Unicode support with DString, you know that most of the non-opaque string code in U++ will use the DString implementation and everything will be biased towards 32 bit.

I prefer 1-3 string classes with an 8 bit bias and the same amount of "String walker" classes and once one implements this, I believe in practice the amount of switching from one encoding to another gets minimized.

I my library I only have String, with plans to add WString someday, and the complexity of Utf8 has never made me wish for DString. Conversion to Utf16 only happens in WinAPI context anyway. Plus on the web, Utf8 is the standard. You will probably find that across the globe, on average, Utf8 is still the smaller method of storage. And in CJK context, Utf16 makes a lot of sense, even with occasional surrogate pairs. Historic/academic CJK is where Utf32 shines, but still, in such contexts, Utf is not the most popular.

Anyway, I would go with the same approach I went so many years ago.

Conversion from Utf8 to Ucs2 must be upgraded to Unicode 9.0 compliant Utf8 to Utf16. Utf8 1-4 code units must be converted to a single code point. Correct escaping and error recovery must be implemented. The Unicode standard give you exact deterministic outcomes for any illformated sequences, and combined with the EE you already have in U++, you can get non-destructive error handling. This will leave you with the ability to read and write Unicode. The rest of U++ won't be helped by this, but it is still an important step and will solve a few problems.

Then, one by one, I would convert pieces of code over to a standardized traversal mechanic, be it a string walker class or something else you come up with.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Tue, 30 May 2017 09:45:16 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 30 May 2017 11:23I meant code point.

Get is the code point.

And what I claimed is that having code point = code unit only half-way solves the problem and you get the least favorite and advantageous encoding in DString/Utf32.

The index of Utf code-point can be easily solved with a string walker that can work for any encoding, from utf8 to 32 and there is no need to tie yourself down with DString supremacy.

Because once you reach "full" Unicode support with DString, you know that most of the non-opaque string code in U++ will use the DString implementation and everything will be biased towards 32 bit.

Interestingly, this is not true at all in current U++ with WString. Most data are always in utf8 String. WString is usually only used temporary to process data. Which was the original intention going to 32 bits 'DString'.

Actually, rethinking graphem problem, maybe we could have

GraphemString

with

String operator[](int i) // returns complete grapheme at position i
int GetCount() // returns number of graphemes

etc...

It would be an 'index' over Utf8 String, grapheme addressed.

See, my problem can be demonstrated by

TextCtrl::Ln

There is the storage for lines of editors. Now 'len' is number of 'cursor characters'. When changes are done to the line, e.g. inserting something at cursor position, text is unpacked to WString, Insert at cursor position is done, then text is packed back to utf8.

Now maybe, if I decide that grapheme is the 'cursor character', going to GraphemeString would be relatively simple. And maybe even faster than using WString...

(All that said, all this will need some way how to provide GUI input / Draw output for graphemes.... but that is a story for another day).

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 08:30:09 GMT
View Forum Message <> Reply to Message

It looks like there are many possible ways to go forward. We can try several things and probably a lot of things will work.

As long as we understand that there is no universal way to make Unicode indexable, but on a

case by case basis, you can. The only thing you can universally do is to iterate linearly over Unicode.

But I still think I gave you a partial solution so many years ago.

To reiterate:
1. Utf8 to Utf16 and vice-versa must be fixed under all scenarios. We also need to add Utf8 to Utf32, but that is trivial compared to Utf16. So proper error recovery must be implemented and 4 byte long sequences must be converted to surrogate pairs.
2. The Unicode table must be expanded to more than 2048 characters. Maybe not full range, but Unicode is based on blocks. We can move a bit closer to the CJK block, because for CJK, nobody expects more than the basics. Probably 8k at least.
3. Take the opportunity to clean stuff up. Core has had several clean-ups and now C++11 shook thing up. So this might be the perfect opportunity to make sure the designs and interfaces are future proof.
4. Implement DString? I don't know yet. On the other hand, implementing DSting is easy and it can be dropped in its own header and available for use.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 09:00:44 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 31 May 2017 10:30It looks like there are many possible ways to go forward. We can try several things and probably a lot of things will work.

As long as we understand that there is no universal way to make Unicode indexable, but on a case by case basis, you can. The only thing you can universally do is to iterate linearly over Unicode.

If you can iterate linearly, Unicode is indexable...

Quote:
But I still think I gave you a partial solution so many years ago.

To reiterate:
1. Utf8 to Utf16 and vice-versa must be fixed under all scenarios. We also need to add Utf8 to Utf32, but that is trivial compared to Utf16. So proper error recovery must be implemented and 4 byte long sequences must be converted to surrogate pairs.

Agreed.

Quote:
2. The Unicode table must be expanded to more than 2048 characters. Maybe not full range, but

Unicode is based on blocks. We can move a bit closer to the CJK block, because for CJK, nobody expects more than the basics. Probably 8k at least.

Not so sure about this - not that important IMO at this point. So I will not get correct ToUpper for many characters - that has little impact on most applications.

Quote:
4. Implement DString? I don't know yet. On the other hand, implementing DSting is easy and it can be dropped in its own header and available for use.

I am now leaning against it. Vector<int> is good enough for utf32 - what we eventually need to do with it.

I am now really thinking that "multibyte" String is the solution. The one that returns a variable sequence of bytes for each position. I am now even thinking this does not need to be bound to graphemes only.

The longterm point with that is to replace WString as processing facility in editors.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 09:30:33 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 31 May 2017 12:00
If you can iterate linearly, Unicode is indexable...

Sorry, you don't get it. I'm using a very specific meaning of indexable, probably that is the problem.

A vector is indexable. You can reach v[7] without going though 0 to 6 and 0 though 6 can't do anything to change the "offset" of 7. A list is not. You need to traverse it to get list[7].

You need to traverse a String s to get to s[7]. The contents up to 7 can change the offset of 7. Unicode strings are not indexable.

Indexable data structures are predictable contant-cost no need to iterate over them to get some "groundind". Non-indexable does not mean that you can't reach them though an index. It means that you can get only obtain that index with a linear traversal O(index) traversal and you need to reach than index at least once.

mirek wrote on Wed, 31 May 2017 12:00
I am now really thinking that "multibyte" String is the solution. The one that returns a variable

sequence of bytes for each position. I am now even thinking this does not need to be bound to graphemes only.

The longterm point with that is to replace WString as processing facility in editors.

That sound to me like you trying to make String indexable. Which can't be done with Unicode. How are you going to determine the sequence of bytes that you must return for a position without traversing it linearly?

If you give it a position as input, this still needs to be a deterministic constant cost memory jump. That should be the bread and butter operation. In the few cases where you need your multibyte behavior, there you abandon indexing randomly into the string (string[7] is forbidden) and iterate over it, going left to right, returning multiple bytes for each logical "position".

Let me sum up the Unicode fallacy:
1. Code points are only indexable in Utf32. Which is not always helpful because
2. Glyps/Graphemes are never indexable.

The common misconception is that 1 and/or 2 are false.

mirek wrote on Wed, 31 May 2017 12:00
I am now leaning against it. Vector<int> is good enough for utf32 - what we eventually need to do with it.

I agree, that is enough. And in some context even C vectors will work.

We do need a standardized functions that convert from code units to a single code point to fill up these structures.

mirek wrote on Wed, 31 May 2017 12:00
Not so sure about this - not that important IMO at this point. So I will not get correct ToUpper for many characters - that has little impact on most applications.

Half measures again...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 10:07:26 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 31 May 2017 11:30mirek wrote on Wed, 31 May 2017 12:00
If you can iterate linearly, Unicode is indexable...

Sorry, you don't get it. I'm using a very specific meaning of indexable, probably that is the

problem.

A vector is indexable. You can reach v[7] without going though 0 to 6 and 0 though 6 can't do anything to change the "offset" of 7. A list is not. You need to traverse it to get list[7].

I guess with "indexable" I was was not specific enough.

Imagine that you traverse through Utf8 String and each codepoint (or grapheme cluster) you store in individual cell of

Vector<String>

Then the result is definitely indexable. Or am I missing something? (Of course, I would not use Vector<String> for optimized version, but that is just implementation issue).

mirek wrote on Wed, 31 May 2017 12:00
That sound to me like you trying to make String indexable. Which can't be done with Unicode. How are you going to determine the sequence of bytes that you must return for a position without traversing it linearly?

Traverse it and store positions. (It is functional equivalent to splitting to Vector<String> as described above).

mirek wrote on Wed, 31 May 2017 12:00
Not so sure about this - not that important IMO at this point. So I will not get correct ToUpper for many characters - that has little impact on most applications.

Half measures again...
[/quote]

I do not mean that it is not important at all. Just that choosing proper basic interfaces is more pressing at the moment.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 10:26:22 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 31 May 2017 13:07
Then the result is definitely indexable. Or am I missing something?

Yes, performance!

Unicode Strings are not indexable and indeed you can make them indexable with Vector<String> (or better).

But that is a bit of Sisyphean act. The conflict between them being non indexable and you forcing them to be indexable will result in performance and memory overhead. Like I said before, you can make a list indexable by traverse and store but you rarely would do this in practice, instead replace your random access algorithm with a linear traversal one if possible.

Now, there are some mighty complex algorithms which probably will call for this, where we will traverse and store.

But for the rest, I still think that traverse and store into a indexable structure is the worst case scenario.

You still traverse the string once, but do not store only the current code point and maybe have a few "last" positions to keep track of some other characters from previous positions.

And I would still advise the use of a StringWalker class, one that can seek to a random position, but ideally the algorithm will never use this capability! After a seek (or just initialization) it will store the current code point and a few more fields, like begin of the code point, size of the sequence. Then it as ++ and -- to go one code point up or down. This class or a separate one can do the same for glyphs.

Using such a class (or embedding this functionality directly into String) to traverse the string once from beginning to end to process each codepoint/glyph will have almost zero performance overhead.

The important part is not to make the confusion that such a class makes string index-able, i.e. only random seek if can't avoid it. And random seek with small jumps. Writing the algorithm in such a way that StringWalker is List<int>, not Vector<int>.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 10:40:19 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 31 May 2017 12:26mirek wrote on Wed, 31 May 2017 13:07
Then the result is definitely indexable. Or am I missing something?

Yes, performance!

No.

Really, the issue at hand is this. Current code works like (inserting character into LineEdit):

```
WString unpacked_line = line[i].ToWString();
unpacked_line.Insert(cursor_position, something);
line[i] = unpacked_line.ToString();
```

I am pretty sure that replacing WString with the new "Indexed unicode" String type will have similar performance, if not better, plus code will not change significantly.

That said, I am really focused at this very issue, this is what I really need to solve.

For most "low-level" tasks like ToUpper walker is better approach.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 11:12:14 GMT
View Forum Message <> Reply to Message

Well since in that example you wont to use some new form of storage and not just a DString, this is more of an implementation detail and not a public API.

I was discussing public API, not LineEdit implementation detail.

I want the parts that are publicly visible from U++ and often used to no longer be a decade behind on Unicode.

As for the LineEdit, there are multiple solutions. I'm pretty sure one can make that work with Utf8 with 1-4 code unit string, but it is a bit more complicated. The only complication is of course getting and changing the cursor for insertion, right? And related to that, but on a side note, my String, when doing find, insert and other such operations, does have multiple variants. You can insert in the middle both a byte like 32 for space or my "Char", which is a DWord, so inserting

everything, code points and code units. You can insert int the middle of a Utf8 string a Utf8/16/32 string, with on the fly conversion, and it helps if the Insert returns the end point of the insertion. You already know the beginning.

For LineEdit, DString, Vector<int> or some multi-byte string would all work I think. I comes down to what one is willing to implement.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 11:20:51 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 31 May 2017 13:12Well since in that example you wont to use some new form of storage and not just a DString, this is more of an implementation detail and not a public API.

I was discussing public API, not LineEdit implementation detail.

But that is just one use of that class. This situation repeats (basically everywhere there is a defined 'position', so requires a tool to handle it. So I am pretty sure it is a useful public api. More useful than WString.

Quote:
For LineEdit, DString, Vector<int> or some multi-byte string would all work I think. I comes down to what one is willing to implement.

I guess we have started with (your correct) argument that DString or Vector<int> are not enough, because we edit characters, not codepoints...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 11:43:50 GMT
View Forum Message <> Reply to Message

Without RLE, I think you can get around this by positions not representing characters, but sequence starts.

As a mandatory condition, every time the position is updated, it is guaranteed to be at the start of a sequence.

As a more complicated example, let's say you have a selection of text, with a "begin" and "end" pos. You handle a key press. Taking your start pos as a sequence start, you determine the sequence end. This means looking to see how many code units it is, seeking over combination marks and ligatures. Basically on the fly glyph analysis. With sequence start end end you know for a fact that everything between these two values must go. You do the same for the end position. As an optimization, you can mark everything for deletion between the start sequence begin and end sequence end. The text marked to be replaced will replaced with multiple code units.

The real challenge is to standardize these operations so you don't have to repeat them.

Maybe we need some GlyphInfoExtractor class or something. Something when given a random sequence of code units and a valid code point start, it can handle such common operations?

Here is a sample from unciode.org:

This is 14 code units, 5 code units, 4 glyphs. The user will see and recognize 4 items as more or less "atomic", so we should focus on this.

We need and API that can locate each glyph start and allow us to replace glyphs 2 and 3 with an

This can be done on the fly with something high level like:

or

or we can go lower level. Or we can go into multi-byte String territory.

PS: the high level stuff still is StringWalker territory.

## File Attachments

1) char_combmark_ex1.png, downloaded 528 times

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 12:41:28 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 31 May 2017 13:43Without RLE, I think you can get around this by positions not representing characters, but sequence starts.

But that is just implementation issue. Semantically, it is the same.

FYI, my new "String" implementation would go like this

```
class MString {
  String text;
  int character_count; // number of code units groups, e.g. graphemes
  union {
    byte *offsets; // for tiny strings
    word *woffsets; // for small strings
    dword *loffsets; // for really big strings
  };

  String operator[](int pos) { // not often used likely, more for demo
    int pos1 = GetOffset(pos);
    return text.Mid(pos1, GetOffset(pos + 1) - pos1);
  }
}
```

Quote:
The real challenge is to standardize these operations so you don't have to repeat them.

Yes. Hence MString. Note that with MString, after you perform Insert of another MString or Remove, you just update offset table, no need to iterate anything again.

Quote:
PS: the high level stuff still is StringWalker territory.

Depends on what how you define "high level" :)

For me, really high level abstraction is where you abstract from those little pesky issues and handle everything in high-level units (graphemes / characters).

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 13:06:35 GMT
View Forum Message <> Reply to Message

OK, let's see it in action.

I still think that MString is a text book case of Unicode indexability fallacy, but maybe I'm wrong.

And if it works, maybe it is fine.

Please let me know if you need some help for the basic stuff. I can also review Unicode conformity of algorithms.

Do you want go full Utf8 minimal valid sequence validation and overlong prevention?

| Code Points | First Byte | Second Byte | Third Byte | Fourth Byte |
|---|---|---|---|---|
| U+0000..U+007F | 00..7F | | | |
| U+0080..U+07FF | C2..DF | 80..BF | | |
| U+0800..U+0FFF | E0 | A0..BF | 80..BF | |
| U+1000..U+CFFF | E1..EC | 80..BF | 80..BF | |
| U+D000..U+D7FF | ED | 80..9F | 80..BF | |
| U+E000..U+FFFF | EE..EF | 80..BF | 80..BF | |
| U+10000..U+3FFFF | F0 | 90..BF | 80..BF | 80..BF |
| U+40000..U+FFFFF | F1..F3 | 80..BF | 80..BF | 80..BF |
| U+100000..U+10FFFF | F4 | 80..8F | 80..BF | 80..BF |

With the rest error escaped?

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 13:25:51 GMT

cbpporter wrote on Wed, 31 May 2017 15:06OK, let's see it in action.

I still think that MString is a text book case of Unicode indexability fallacy, but maybe I'm wrong.

And if it works, maybe it is fine.

Please let me know if you need some help for the basic stuff. I can also review Unicode conformity of algorithms.

Do you want go full Utf8 minimal valid sequence validation and overlong prevention?

| Code Points | First Byte | Second Byte | Third Byte | Fourth Byte |
|---|---|---|---|---|
| U+0000..U+007F | 00..7F | | | |
| U+0080..U+07FF | C2..DF | 80..BF | | |
| U+0800..U+0FFF | E0 | A0..BF | 80..BF | |
| U+1000..U+CFFF | E1..EC | 80..BF | 80..BF | |
| U+D000..U+D7FF | ED | 80..9F | 80..BF | |
| U+E000..U+FFFF | EE..EF | 80..BF | 80..BF | |
| U+10000..U+3FFFF | F0 | 90..BF | 80..BF | 80..BF |
| U+40000..U+FFFFF | F1..F3 | 80..BF | 80..BF | 80..BF |
| U+100000..U+10FFFF | F4 | 80..8F | 80..BF | 80..BF |

With the rest error escaped?

Not sure about that, but what I know for sure that I want to put decoding in single template (unlike current charset.cpp) so that it can be fixed easily...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 13:34:02 GMT


OK, after thinking about it, overlong should be error escaped.

I think that the basic routine should produce some error flag if it does error escape.

I also think that (maybe on flag), I would like the basic encoding extended to full 32-bits (with error flag). The reason is that it could be handy outside character use (e.g. storing relative offsets of something).

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 31 May 2017 13:38:48 GMT

View Forum Message <> Reply to Message

mirek wrote on Wed, 31 May 2017 16:34
I also think that (maybe on flag), I would like the basic encoding extended to full 32-bits (with error flag). The reason is that it could be handy outside character use (e.g. storing relative offsets of something).

Sorry, I do not understand what you mean here...

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 31 May 2017 13:50:12 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 31 May 2017 15:38mirek wrote on Wed, 31 May 2017 16:34
I also think that (maybe on flag), I would like the basic encoding extended to full 32-bits (with error flag). The reason is that it could be handy outside character use (e.g. storing relative offsets of something).

Sorry, I do not understand what you mean here...

Ah, it is not really unicode related. But sometimes you have a set of 32-bit (or 31-bit) numbers where you know that most of them are <128 (but some are not) and want to store it effectively. So I was thinking that it would be nice to reuse the very same code for this...

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 05 Jun 2017 15:51:46 GMT
View Forum Message <> Reply to Message

Utf[8,16,32] <=> Utf[8,16,32] conversion routines commited...

Any tips for good unicode classification data (in single file)?

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 06 Jun 2017 07:28:38 GMT
View Forum Message <> Reply to Message

You mean Unicode General Category?

I have that encoded in my two table scheme.

If that is all you can want to include, it should be extremely small, something like (number of characters / chunk size) * 1.2.

If you mean scripts, that is a bit more complex, mostly because I want the data small.

I'm also working on Unicode names. It turns out that there is under 200 unique words in the names. I want to use a huge string buffer where each byte in the string is a word, not a character.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Tue, 06 Jun 2017 08:41:27 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 06 June 2017 09:28You mean Unicode General Category?

I need to make some sense of it all... :)

Not yet sure what exactly I will need, but for now I am pretty sure I would like to have info that e.g.

is character based on

C

with

combining character.

That C is uppercase and there is corresponding lowercase c. Now I can se I can have

"Latin Capital Letter C with caron"

which is probably OK, but not sure if it is without ambiguities.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 06 Jun 2017 09:18:44 GMT
View Forum Message <> Reply to Message

There is no easy solution here I'm afraid.

You probably know the file:
http://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt

What I do is read that file, compile all the information in RAM and write out C++ tables.

The file has a lot but not all of the needed information. The question is how much of it you need and how are you going to store it.

I don't understand the final point you were making about ambiguities? Characters are uniquely defined, so are the canonical composition and decomposition rules, together with compatibility substitutions.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Tue, 06 Jun 2017 11:21:34 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 06 June 2017 11:18
Characters are uniquely defined, so are the canonical composition and decomposition rules, together with compatibility substitutions.

I suppose so. Still learning.

Your lib is BSD? Available somewhere?

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 06 Jun 2017 11:39:19 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 06 June 2017 14:21cbpporter wrote on Tue, 06 June 2017 11:18
Characters are uniquely defined, so are the canonical composition and decomposition rules, together with compatibility substitutions.

I suppose so. Still learning.

Your lib is BSD? Available somewhere?
Oh yeah, the Unicode spec is huge. But it also recommends that you implement as much as you need for your needs, not the whole thing.

The lib is a bit more complicated. It is Apache Version 2.0, but not really released yet. More precisely, the more advanced Unicode parts only exist on my disks yet, they are not committed. But when they will be committed, it will be under Apache. How does future licensing work? :)

But I can share that, with the only caveat that the only things I'm really doing and are meant for the final lib is conversion and encoding of UnicodeData.txt. I'm not planning on composition handling for now since it is not about GUI or displaying text. My encoding scheme handles 4 things: category, upper, lower and title case.

I would like still to add script to that data and probably you can't dodge forever adding canonical normalization support.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 06 Jun 2017 11:58:53 GMT
View Forum Message <> Reply to Message

As an example, here is my extractor.

It is part of the workbench projects, small poorly written fire and forget programs, so expect it to be messy. Super messy and not maintained.

This one generates two tables which are then used by the real code to power 3 Unicode plane functionality.

File Attachments
1) udb.zip, downloaded 236 times

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Thu, 08 Jun 2017 08:00:04 GMT
View Forum Message <> Reply to Message

My priority is CodeEditor right now, but right after I do want to take care on my side of canonical decomposition too.

I'll gather stats like what % of characters can be decomposed and into how many characters on average to come up with an optimal scheme. For Latin languages I expect a few hundred with at most 3 characters in decomposition, with most having 2.

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Thu, 08 Jun 2017 08:26:47 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 08 June 2017 10:00My priority is CodeEditor right now, but right after I do want to take care on my side of canonical decomposition too.

What do you plan?

Quote:
I'll gather stats like what % of characters can be decomposed and into how many characters on average to come up with an optimal scheme. For Latin languages I expect a few hundred with at most 3 characters in decomposition, with most having 2.

That is my estimate too. I even thing that 3 codepoints is so sparse, that the basic table should only store 2 (which means single base char + single combining mark) - that will allow for more dense table, and 3 codepoint characters should be handled as exception.

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Thu, 08 Jun 2017 08:43:00 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 08 June 2017 11:26cbpporter wrote on Thu, 08 June 2017 10:00My priority is CodeEditor right now, but right after I do want to take care on my side of canonical decomposition too.

What do you plan?

I guess you missed my saga: http://www.ultimatepp.org/forums/index.php?t=msg&th=9945&start=0&

I'm guessing I have about 3-4 hours more of work and I'll have a preview version done. Then I'll upload it there and include it into my daily builds and test it for a couple of weeks.

mirek wrote on Thu, 08 June 2017 11:26
Quote:
I'll gather stats like what % of characters can be decomposed and into how many characters on average to come up with an optimal scheme. For Latin languages I expect a few hundred with at most 3 characters in decomposition, with most having 2.

That is my estimate too. I even thing that 3 codepoints is so sparse, that the basic table should only store 2 (which means single base char + single combining mark) - that will allow for more dense table, and 3 codepoint characters should be handled as exception.

That's why I'm gathering data to make informed decisions. I'll get back to you with the stats.

How do you want to handle compatibility decomposition? Like:
http://www.fileformat.info/info/unicode/char/0149/index.htm

My plan is to have a flag for compatibility decomposition vs normal ones, with it being off by default. I'm not sure, but I think you can exclude them all if you don't want to bother with. Unicode can be complicated.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Thu, 08 Jun 2017 09:22:53 GMT
View Forum Message <> Reply to Message

Here are the primary stats:
- there are only 5721 characters that have decomposition.
- 2060 out of them have normal decomposition. All of these are two characters. So representing them is easy. Unfortunately the highest CP is 2FA1D. But it is CJK compatibility. I think it should be ignored. Without those the highest codepoint is 1D1C0. But if you ignore a bunch of stuff, like

hebrew, hiragana, musical notations I think one could stop even as low as the aptly named character: http://www.fileformat.info/info/unicode/char/2adc/index.htm

Going lower than 0x2adc will soon cut of stuff like Greek. You need 2000 to not cut off Greek.

- the rest of decomposition are 2 to 4, but there are some weird exceptions, like a 18 character one.
- if you stop ar 0x2000/Greek the max CP a decomposition is 8190. If yous top at 0x2adc, it is 12297.

So a dump scheme for the first 0x2adc or a bit higher would be 43.888 bytes. In my lib I already have (256 * 256 * PLANES / BS) * 2 + count2 * 8 = 131456 bytes of data used by uppercase/lower case, so even 43k more is pushing it. I'll investigate how to represent sparsely both the first 0x2adc characters with exact 2 character long decomposition the entire Unicode range.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Thu, 08 Jun 2017 11:00:01 GMT
View Forum Message <> Reply to Message

Not quite 100% done analyzing the data, but here is what I think I'll do:
- respect the 3 plane convention. Unicode has 17 planes, with the first 3 in active use. Plane 14 is used, but it is specific and only has 368 allocated code points. It is so specific that I'll add exclude it, the same as I do all planes except planes 0-2. All excluded planes have the property that any function f(cp) = cp.
- I'll ignore all special substitutions: sub and superscript, font, circle, square, fractions and of course compatibility substitutions. I won't be using a flag for now, just exclude them.
- I'll ignore all CJK COMPATIBILITY IDEOGRAPHs. There is no way a general purpose library can provide satisfactory use case for these. If you really needs such substitution, you will probably use a more competent third party library. f(CJK COMPATIBILITY IDEOGRAPH) = CJK COMPATIBILITY IDEOGRAPH

All these combined with my two table solution, with a chunk size of 256 to 1024 will leave me with around 8000-9000 bytes of data in each executable that does decomposition. Final numbers will be determined once implementation is done and round trip testing is complete.

I think this is a reasonable subset that can handle NFD, at the small price of a flat 9K in exe size, + the size of the actual methods.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Sun, 11 Jun 2017 11:57:38 GMT

cbpporter wrote on Thu, 08 June 2017 13:00Not quite 100% done analyzing the data, but here is what I think I'll do:
- respect the 3 plane convention. Unicode has 17 planes, with the first 3 in active use. Plane 14 is used, but it is specific and only has 368 allocated code points. It is so specific that I'll add exclude it, the same as I do all planes except planes 0-2. All excluded planes have the property that any function f(cp) = cp.
- I'll ignore all special substitutions: sub and superscript, font, circle, square, fractions and of course compatibility substitutions. I won't be using a flag for now, just exclude them.
- I'll ignore all CJK COMPATIBILITY IDEOGRAPHs. There is no way a general purpose library can provide satisfactory use case for these. If you really needs such substitution, you will probably use a more competent third party library. f(CJK COMPATIBILITY IDEOGRAPH) = CJK COMPATIBILITY IDEOGRAPH

All these combined with my two table solution, with a chunk size of 256 to 1024 will leave me with around 8000-9000 bytes of data in each executable that does decomposition. Final numbers will be determined once implementation is done and round trip testing is complete.

I think this is a reasonable subset that can handle NFD, at the small price of a flat 9K in exe size, + the size of the actual methods.

I have managed to squeeze complete composition to 3.8KB table... :)

Interesting observation: With UnicodeCompose / Decompose, with first 2048 codepoints covered by "fast table", there is no need for further tables for ToUpper, ToLower, ToAscii.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 07:39:42 GMT


What? How? Need to check it out.

I made a mistake in the estimate and actually how much data I need and the 9k got up to 34. So I started working on a 3 table solution :).

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 12 Jun 2017 08:13:44 GMT


cbpporter wrote on Mon, 12 June 2017 09:39What? How? Need to check it out.

I made a mistake in the estimate and actually how much data I need and the 9k got up to 34. So I started working on a 3 table solution :).

Really trivial. 4 Vectors of dwords (original code, 3 decomposed codes), then delta it (that will result in most values being the same), then ZCompress... (that will RLE and Huffman those same values).

That said, I am only doing decomposition of characters in UnicodeData.txt.

Also, I perhaps need to add this

https://en.wikipedia.org/wiki/Korean_language_and_computers# Hangul_in_Unicode

too...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 08:21:40 GMT
View Forum Message <> Reply to Message

If you use an up to date UnicodeData or the one I uploaded and you go though the entire file, you should have 100% coverage of all 120k Unicode codepoints. But do take care to cover the gaps.

I was thinking about calling compress too, but I'm not sure. That's why I cam up with the table scheme. Even if I manage to massively compress it, I don't want three planes of Unicode eating up a ton of RAM with a "dumb" decompressed massive memory layout.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 12 Jun 2017 08:28:12 GMT
View Forum Message <> Reply to Message

It does not need to. I am using Indexes after decompression.

Frankly I yet need to calculate how much memory it takes, but I think it will be about 100K.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 12 Jun 2017 08:31:33 GMT
View Forum Message <> Reply to Message

120K of Index data. Entirely acceptable for me.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 08:53:27 GMT
View Forum Message <> Reply to Message

For how many code points do you have upper/lower support?

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 12 Jun 2017 08:57:51 GMT
View Forum Message <> Reply to Message

It is all work in progress, at this moment, I have only that composition/decomposition.

However, as I have written before, if you cover first 2048 codepoints with separate 'fast' table (which I plan to do anyway), it is possible to implement lowercase/uppercase just by decompose, alter first codepoint (using 'fast' table) and then recompose. Seems to work for all codepoints > 2048.

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 09:37:22 GMT
View Forum Message <> Reply to Message

Interesting things in uppbox.

I might be able to reduce my 130 Kib UnicodeData.

But dammit, this is not what I'm supposed to be doing right now! You managed to side track me :).

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 12 Jun 2017 09:41:42 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 12 June 2017 11:37Interesting things in uppbox.

I might be able to reduce my 130 Kib UnicodeData.

But dammit, this is not what I'm supposed to be doing right now! You managed to side track me :).

...or you can wait for me to finish... :)

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 10:50:54 GMT
View Forum Message <> Reply to Message

No, I'll do it too. Reducing the 130 KiB will be welcome but really not a priority. If I had time, I would do it right now.

But the CodeEditor reword is a bust for now. There is no easy way to map a CodeEditor to all the CSyntax objects that are created and retroactively update them. Plus I represent syntax in expensive to copy structures, so I need to rework that. But scheduling is not on my side. So for some time more I'll continue using the CodeEdtior fork.

Plus, I spent most of the day today first isolating Pdb from ide/Debuggers and then from IDE. It almost compiles. But I have one major problem left before it compiles:

I can't find AK_ADDWATCH.

I searched all of U++ and WinSDK and I couldn't find where it is defined.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 11:06:16 GMT
View Forum Message <> Reply to Message

Found it!

Shenanigans again with those include file tricks you like :).

Anyway, it compiles and links. Crashes on start even though nothing is called. But I'll get it to work!

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 12 Jun 2017 12:20:51 GMT
View Forum Message <> Reply to Message

Back to composition: tested out the 3 table method. Can't get it under 24000 bytes. I need to represent 2116 code points and that is almost 17000 bytes. The rest is index data.

But seeing as there are only 7189 * 8 bytes of case data represented as 130 KiB, I guess it would be better to touch up on that and leave composition as is. There must be a better way to compactly represent 7189 code points from 200k ones. That is super sparse.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Tue, 13 Jun 2017 14:31:09 GMT
View Forum Message <> Reply to Message

Well this was quite frankly not necessary and a huge waste of time, but I managed to get down my Unicode data from 130K to 68K. It includes 3 planes with charter type, upper, lower and title case. I guess the nonexistent users of my library will be happy :).

I should have probably went with your compressed scheme, but I'm stubborn. We'll see what the future holds, since only now am I getting to writing the decomposition API.

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 14 Jun 2017 09:07:30 GMT
View Forum Message <> Reply to Message

Why the hell am I worrying about size of executables? It is not like hello world is not 400 KiB under TDM :).

Anyway, Mirek, please let me know when the new support is in in Core in a nightly.

There is pretty much one way to convert from Utf8 to Ut16 and co, so comparing codes is a pretty good method of spotting errors.

Thank you!

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 14 Jun 2017 10:07:55 GMT
View Forum Message <> Reply to Message

It looks like decomposition is not as simple as I first though. Take a look at:
http://www.fileformat.info/info/unicode/char/1f80/index.htm

This character decomposes into a composed character and a mark. So I guess decomposition needs to be recursive.

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 14 Jun 2017 10:17:32 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 14 June 2017 11:07Why the hell am I worrying about size of executables? It is not like hello world is not 400 KiB under TDM :).

Auhm, I still do, to the extent. Surely 10KB is nothing, but 500KB would be too much - just for unicode support that 99% of users is never going to use.

Quote:
Anyway, Mirek, please let me know when the new support is in in Core in a nightly.

There is pretty much one way to convert from Utf8 to Ut16 and co, so comparing codes is a pretty good method of spotting errors.

Thank you!

What I have implemented is already in trunk Core (and that means in nightly).

A lot is missing. I want to update existing 'fast' tables - I have some new about the information that I would like to know, like

- IsRTL
- IsWide
- IsSymbol
- IsControl

etc...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 14 Jun 2017 10:30:49 GMT
View Forum Message <> Reply to Message

Good find. What do you suggest to do about that?

- I can leave current code as is and perhaps add "FullDecompose" variant.
- I can decompose it to 3 codepoints outright

I think I like the second option better...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 14 Jun 2017 10:42:01 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 14 June 2017 13:30Good find. What do you suggest to do about that?

- I can leave current code as is and perhaps add "FullDecompose" variant.
- I can decompose it to 3 codepoints outright

I think I like the second option better...

Here is what I am implementing right now:
1. A Decompose method. You give it a code point and it gives you raw UnicodeData.txt data. That problematic character will still give you two results. This is already done.

But I practice I doubt it will ever be used, so much so that it barely qualifies as a public method. Instead, everybody will use...

2. A ToNFD() method. NFD is the canonical decomposition. The problematic character will result in 3 code points. This will be the main public method.

So my main method will be the method you prefer, giving 3 results. I just gave it the Unicode name.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Wed, 14 Jun 2017 17:09:30 GMT
View Forum Message <> Reply to Message

even more fun is:

3311

That expands to two katakana characters and mark, and both katakana characters further expand to base katakana and mark.

All in all expands to 5 codepoints.

Trouble is that recompose must account for all combinations. I guess it will have to be multipass...

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 14 Jun 2017 21:19:17 GMT
View Forum Message <> Reply to Message

I hope the NDF algorithm chapter on unicode.org covers that: the hows and whys.

The whole thing is pretty crazy though. I'll have excellent Unicode support eventually, but there is no way to get it under 100k data.

But I did experiment with Zlib, and all the tables can be squashed down a ton. Except the case table, which only goes down to 50%. The only problem is that I don't have any Zlib support in the my library yet. Plus, I would like to add conditional compilation.

You inspired me with the plugin system. I would like uncompressed data if the z plugin is absent, otherwise automatic compression. I tested that the exe size growth due to zlib is outweighed by the table compression. Deflate is pretty small.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Wed, 14 Jun 2017 21:31:09 GMT
View Forum Message <> Reply to Message

PS: that is a special composition. I went over the data over and over again and I found no good

reason to handle box decomposition.

It is not like U++ will check to see if the font supports that character and if not, decompose it and build CJK in a small box on the fly.

Decompositions that start with <smth> are all special, like <font>, meaning that you can decompose that character if you are doing font substitution to an approximation or <square>, meaning that the code point is multiple characters arranged in a square or like <fraction>, when you have 1/2 as a single code point and you can decompose it as 1/2, using 3 code points.

I choose to ignore all these for now since I can't figure out how to offer any worthwhile feature related to these special substitutions. I don't even need normal decompositions, but it is pretty cool to decompose diacritics and replace some bits since I'm an European and my native language uses diacritics.

As for NCF and NDF I only found two good use cases: string equality and search. With the forms, you don't compare code points, but glyphs, without building glyphs. If two string look the same on your display, but have different code points due to diacritics, it is very useful to tell if they are

encoded as a "t" with a composition mark to be identified as the same string.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 19 Jun 2017 08:03:40 GMT
View Forum Message <> Reply to Message

My undestanding is that if decomposition sequence starts with "<", it is 'compatibility', if not, it is 'canonical'.

I believe that you should use compatibility seqences e.g. for comparing, but you should never 'recompose' these into single codepoint - one of reasons is that canonical compositions are unique, but there can be the same compatibility decompositions for multiple codepoints (found out that hard way during testing).

In either case, i have added a bool

int  UnicodeDecompose(dword codepoint, dword t[MAX_DECOMPOSED], bool& canonical);

to 'decompose' API and Compose is now not using noncanonical decompositions.

I believe that my "Unicode INFO" code is now complete. In the end, it is about 12KB of data (6KB compressed and 6KB of 'fast tables' for the first 2048 codepoints).

Documentation needs updating. Then the next part would be updating / deprecating those ToLower/ToUpper routines for Strings, and most importantly, implementing "apparent character logic".

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 19 Jun 2017 08:22:08 GMT
View Forum Message <> Reply to Message

mirek wrote on Mon, 19 June 2017 11:03My undestanding is that if decomposition sequence starts with "<", it is 'compatibility', if not, it is 'canonical'.

I believe that you should use compatibility seqences e.g. for comparing, but you should never 'recompose' these into single codepoint - one of reasons is that canonical compositions are unique, but there can be the same compatibility decompositions for multiple codepoints (found out that hard way during testing).

That's why you read the spec!

Everything is convention based.

Compatibility decomposition and everything that is marked in compatibility in Unicode means that it would not be part of Unicode and has no reason to exist in a standalone standard, but it had to be added to be compatible with another standard.

Canonical is the only that is needed for comparing and search.

Compatibility decomposition and non-compatibility decomposition are separate entities with separate names and compatibility one should not be used unless you are trying to be compatible with another standard.

And the rest can be ignored. Like <font> substitutions:
http://www.fileformat.info/info/unicode/char/2102/index.htm

I really don't think that users expect that hollowed out C to return true when compared to a plain C.

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 19 Jun 2017 08:23:46 GMT
View Forum Message <> Reply to Message

int UnicodeDecompose(dword codepoint, dword t[MAX_DECOMPOSED], bool& canonical);

PS: Unicode doesn't and shouldn't tell you if the decomposition is canonical or not. You ask it for one or the other, never both in the same string.

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 19 Jun 2017 08:40:02 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 19 June 2017 10:22

I really don't think that users expect that hollowed out C to return true when compared to a plain C.

I think this is where we differ.

I really believe that if I, as user, have a long document and I am searching for "Come", I really want to find it the one starting with hollowed C too. Or, e.g. FB01 you want to match upon searching for "fi".

(not that I am going to implement that anytime soon, but IMO this is the expected behaviour).

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 19 Jun 2017 08:51:35 GMT
View Forum Message <> Reply to Message

Yeah, that's why those decompositions are special and should not be used in day to day use.

Canonical decomposition in Unicode is not something to convert to when needed to solve some task. It can be used as the sole encoding format for all your string because it doesn't change the meaning of the string. Example: LoadFile("c:\\utf8.txt"). You could return the string as is, or you could return one of the two canonical Unicode forms, with the original string never stored.

On the other hand, <font>, <box> and other non standard decompositions change the meaning of the text. They are computed when needed and you can't store your string as such 100% of the time.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by mirek on Mon, 19 Jun 2017 08:58:41 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 19 June 2017 10:51Yeah, that's why those decompositions are special and should not be used in day to day use.

Searching documents is day to day use. I think you got this wrong:

Quote:
Compatibility decomposition and everything that is marked in compatibility in Unicode means that it would not be part of Unicode and has no reason to exist in a standalone standard, but it had to be added to be compatible with another standard.

Canonical is the only that is needed for comparing and search.

Compatibility decomposition and non-compatibility decomposition are separate entities with separate names and compatibility one should not be used unless you are trying to be compatible

with another standard.

It really is not about other standard (well, can be, but not only). It is about equivalence when searching.

---

Subject: Re: Choosing the best way to go full UNICODE
Posted by cbpporter on Mon, 19 Jun 2017 09:07:37 GMT
View Forum Message <> Reply to Message

Day to day use: standard encoding scheme, i.e. guaranteed form to have a string in.

Reading from a no normalized stream and getting NFD or NFC is not an error and can be a standard behavior in a library.

Not day to day use: on demand encoding scheme.

Reading from a non normalized stream and getting <font> substitutions is an error. Taking a string and converting it on the fly to a <font> substitution, doing a search to find that "C" like in the example and then discarding the string is not an error.

There is no way I'm getting Unicode wrong.

The only misunderstandings are related to my use of English or being to vague in my expression :).

---