Delusions Of Grandeur

by Julian Bucknall

Perhaps you've never thought about it. Well, it's about time you did: it's a big wide multilingual world out there. What am I talking about? In this particular case, how to compare or sort strings, that's what. A topic that's turns out to be like peeling an onion, there's always another layer underneath.

A Different Point Of View

A quick test to get us going: in what alphabetic order would you put the words rascal, résumé, and ripple? Most English-speaking people, including myself, would put them in the order shown, which is arguably the standard way (at least that's the way my dictionary does it).

So, now a more difficult test, at least to do off the top of your head. Suppose you wrote a sorting routine which sorted string arrays and you used the Delphi < or > string operator to compare two strings as part of that sorting algorithm. What would the final order be if you fed it an array containing our three test words? And if you were clever and used either the `CompareStr` or `AnsiCompareStr` routines from `SysUtils` instead?

I wrote a small program (which I won't reproduce here) that did exactly that: sorting our three test strings with the three methods. The results were interesting. The standard (< or >) operator sorted the strings in the order rascal, ripple, résumé, as did the `CompareStr` routine. The `AnsiCompareStr` routine got it 'right.'

The reason for the difference lies in the implementation of the three routines. The first two rely on a binary comparison: take each character in turn from each string, from the first to the last of the shorter of the two strings, typecast it as a byte and then compare these bytes. Once you get an inequality between the bytes you have an inequality between the original strings. If you don't have an inequality and the strings are equal in length, you have equal strings. Pretty simple, huh? The other routine uses the Windows API routine `CompareString`. `CompareString` uses the 'meaning' behind each character to do its comparison: ie, é should be sorted between e and f.

Another interesting thing happened when I wrote my little test program. I don't know about you, but when I write little test programs to check an algorithm or idea, I use console apps. I can't be bothered dropping components on forms and setting `OnClick` event handlers and label captions and the like. I go straight to the good old `WriteLn` routine in a console app. Except... when I ran the app, the word résumé appeared with the é characters replaced by the Greek theta character. The first time this happens it can be a little disconcerting.

And so my original simple experiment starts getting a little wild. Where did those thetas (thetae?) come from? The answer is that Windows and the DOS box (or the console, if you like) use different code pages to display their characters.

To Step Aside

The term code page is perhaps a little hard to relate to, it's probably better to use the term *character set*. A character set is a set of encodings of numeric values to represent specific characters. In our discussion here we'll be talking about byte encodings; when a word value is used instead it's called Unicode. Hence, for example, the character 'A' is represented by the value 65 (`$41`), 'B' by 66 (`$42`), '0' by 48 (`$30`), etc.

Wait, I hear you cry, those are just ASCII values, what's so special about that? Correct, but standard ASCII only specifies a 7-bit encoding of characters, ie, 128 characters in all. A byte encoding can have 256 values, therefore ASCII is 'missing' 128 characters. When Microsoft and IBM came out with the original MSDOS and PC, they defined an extra set of 128 characters to be used on top of the original 128 ASCII ones (it was known as Extended ASCII). They decided to use a mixture of characters with diacritical marks (accents and the like), line-drawing characters, Greek alphabet characters and so on. As it happened, this was fine for the USA, but the rest of the world wanted extra characters to fill out their alphabets: other characters, characters with special diacritical marks and so on. I well remember the days in England when your IBM PC displayed the pound sterling character (£) but it was printed on your Epson dot matrix printer as something else entirely (IBM and Epson disagreeing on the definition of the Extended character set), there used to be whole magazine articles on how to get things to print properly.

With MS-DOS 3.0, Microsoft came out with COUNTRY.SYS and DISPLAY.SYS and code pages and special display font files and suddenly other countries' citizens could see their own alphabet on the PC screen. Each of these code pages (or character sets) used the same ASCII characters for the first 128 encodings, but the second 128 encodings were country-specific. These different character sets were distinguished by a numeric value, the code page number. For example the original PC code page was number 437, the one used in most of Western Europe was 850, Slavic countries used 852 (it has all those amazing accents that enable you to type the name of the composer of the New World Symphony).

Windows uses two character sets: the ANSI character set (generally 1252 in the West) and the OEM character set (this depends on how your system was set up, mine for instance is 437 at home, but 850

at work). The OEM character set is used for DOS programs, console programs, file names and the like whereas the ANSI character set is used everywhere else. I'm typing this article with Microsoft Word on Windows 95 and it's using the ANSI character set.

You can experiment with the differences in the two character sets from your keyboard if you like. Start NotePad as one window and an MSDOS prompt in another. We'll use the `Alt` key with the numeric keypad.

In NotePad, press the `Alt` key and, holding it down, type `0156`. Release the `Alt` key. Now, on my system (code page 1252), that'll give you the œ ligature. Switch to the MSDOS window and do the same at the DOS prompt. For me (code page 437), I get the pound sterling sign (£). Experiment a little. If you use values less than 128 you tend to get the same character displayed in both windows (try and keep away from `Alt+000` to `Alt+031`, though), but if you use values above 128 you will not get the same character. For me, `Alt+0233` produces either é in Windows or theta in DOS (é in code pages 437 and 850 is `Alt+0130`).

To Face The Truth

So, having made this little diversion into code pages we see that sorting text strings can be somewhat complicated.

Before we can compare two strings the first thing we need to know is which code page they're encoded with. We cannot compare two strings that are encoded differently: one of them must be converted into the other's code page first. Mind you, generally the strings we compare are always in the same code page, and so we shall leave the subject of code page conversion to another day.

Next we need to know if we are doing a case-insensitive comparison or a case-sensitive one. In the former case, lowercase letters are deemed to compare equal to their uppercase equivalents, in the latter the lowercase letters are deemed (usually) to compare greater than uppercase.

Now we can compare the strings. But how? Since é is supposed to be between e and f, and yet the ASCII values of the three characters are 233, 101 and 102 respectively (at least in code page 1252), we obviously cannot use the encoding value as a basis for comparison. In our example we need the result 101, 233, 102, but if we compare ASCII values we'll get 101, 102, 233. And this, by the way, is the problem with Delphi's > and < string operators and the `CompareStr` routine: they treat characters as their byte equivalents.

This is one place where the `AnsiCompareStr` routine comes into its own. It uses the Windows `CompareString` routine which knows how to compare two text strings 'properly.' Its downfall is that it uses the ANSI code page only, and thus depends on how the system has been set up. The `CompareString` that my Windows 95 installs will be different to one in the Czech Republic, for example.

What we need to do is take control of our string comparisons. We need to provide another encoding which says, in effect, the character 101 (e) must be less than the character 233 (é) which, in turn must be less than the character 102 (f). Let's call this encoding the sort order or collation.

What we do is to assign a sort value to each character using an increasing sequence and then all we need to do is sort these sort values. So, for example, if the letter e was assigned the sort value 42, we could assign é the sort value 43 and f the sort value 44. Bingo, e compares less than é which compares less than f, just because 41 < 42 < 43.

Metamorphosis

This, then is our plan. To compare two text strings, we convert each string of characters to a string of sort values and then compare these two converted strings. For simplicity sake, the conversion process uses an array of sort values, so we could write:

```
for I := 1 to length(S) do
  ConvertedS[I] :=  SortValue[S[I]];
```

where `S` is the original string, `ConvertedS` is the string of sort values, and `SortValue` is the collation array. This can be coded in assembly language very efficiently. Of course, an alternative would be to convert each character in turn, one from each string, and then compare the sort values until you reach an unequal comparison.

In fact, if you think about it, we could also provide a collation array for case-insensitive comparisons: just give the lowercase letters the same sort value as the uppercase ones. Suddenly we have an extremely flexible and extensible method of comparing strings. Instead of having a bunch of flags or different routines (for example, do case insensitive comparison? Set flag `X` to `True`. Use ASCII value comparison? Use `CompareStr`) we can supply a collation array to a single routine and away we go.

At this point we could design the comparison routine to look like:

```
function CompareStrings(
  S1 : string; S2 : string;
  Collation : TCollation) : integer;
```

where `TCollation` is an appropriate type for the array of sort values, and the return value is less than zero if `S1` compares less than `S2`, zero if they compare equal, greater than zero otherwise. The actual instances of `TCollation` that you use might come from a file or files, from resources or whatever. Do note however that two strings that compare equal are not necessarily the same string: in a case-insensitive comparison 'ABC' compares equal to 'abc' for example, although they obviously are not the same string at all.

Yesterday, When I Was Mad

Great! At this point you might be reaching for the magazine diskette to unzip the source code and start patching it into your latest application. But, stay awhile, this article is not yet over.

I warned you before that the subject of string comparison just presents more layers the more you pick at it. Those of you who are interested in type (and by that I mean

the form of the letters you see on a page, the font) may have noticed a throwaway reference above to a ligature. I you aren't interested in type you might have glossed over the word. So, what's a ligature and what does it have to do with us?

My youngest cat's name is Aristæus, 'Arry for short. The æ character pair in his name is a ligature: a letter formed by squeezing two or more 'standard' letters together. The other standard ligature is the œ pair, however in some specialist fonts you will also find the ligatures fi, fl, ffi, ff and maybe others. These specialist typographical ligatures are not of concern to us here since they are not characters in standard code pages.

Back to comparing strings: so what's so special about ligatures? Well, does Aristæus compare equal to Aristaeus? Answer: um, yes, I suppose it does. If we were in Germany, would straße (a street) compare equal to strasse? Er, well, yes, the sharp-s character is equal to a double s. Are you getting the gist of this particular problem? Certain single characters must be counted equal (to be strict, compared equal) to a character pair. In other words we must make the æ character have the same sort value as the ae character pair, or the ß character have the same sort value as the ss character pair.

The way we do this is to convert the single character into its equivalent character pair before performing the comparison. As an aside, note that this means a string of characters may grow in size when converted to a string of sort values.

Are we finished yet? Ha, no, of course not. Let's visit Spain briefly. There the character pair ll is counted as a special character that sorts between l and m. Yes, a pair of ls is seen as a single character with a y-type sound and it comes after l and before m. In Spanish dictionaries, llaga (an ulcer, what we're getting here!) will appear after all words beginning with a single l (eg, luna, the moon, the root of lunacy) and before all words beginning with m (eg, macaco, a monkey; no comment). In this case we need to convert the ll pair into a single character, the opposite of the ligature case.

It's Alright

After identifying these two slightly different problems we finally come to the end of our particular quest. We need to enhance our posited `TCollation` type to include this "1-for-2" and "2-for-1" information as well. We need to be able to read a collation from a file or, even better, a stream at run-time (entering a collation to be compiled into the EXE is not my idea of fun).

Of course, what I am trying to get at here is that ideally a `TCollation` class needs to be defined, not the simple record structure which we posited before. Its methods will read in the collation data from a stream, compare two strings, convert a string to its sort value, and so on.

I won't reprint the source code here; it's all pretty simple and, in doing so, this already long article won't be particularly enhanced. Go, look at the source.

After all, my point in writing this article is to make you realize that there is more to comparing strings than using the Delphi string operators. If, every time you compare two strings, a little voice in your head starts squeaking "code page," "collation," "ligature," then this article will have served its purpose. Learn your lesson from me: this code was written to be part of TurboPower's FlashFiler product after some of our European customers kindly pointed out that their string keys were not being indexed properly.

Opportunities

The code that accompanies this article is for Delphi 2 and 3 only. It defines a class that has method to compare two strings, a method to convert a string to its collation equivalent (called a sort string in the code) and so on. There is also a `TStringList` look-alike class that sorts its strings according to a collation you supply. And to help out there are several files defining different collations for different code pages, for case-insensitivity, for different countries and so on. Please see the COLLFILE.TXT file for a description of the different collations available.

Julian Bucknall is Director of Tools Development at TurboPower Software. He can be reached by email at julianb@turbopower.com or on CompuServe at 100116,1572