Characters and Grapheme Clusters

It's common to think of a string as a sequence of characters, but when working with `NSString` objects, or with Unicode strings in general, in most cases it is better to deal with substrings rather than with individual characters. The reason for this is that what the user perceives as a character in text may in many cases be represented by multiple characters in the string. `NSString` has a large inventory of methods for properly handling Unicode strings, which in general make Unicode compliance easy, but there are a few precautions you should observe.

`NSString` objects are conceptually UTF-16 with platform endianness. That doesn't necessarily imply anything about their internal storage mechanism; what it means is that `NSString` lengths, character indexes, and ranges are expressed in terms of UTF-16 units, and that the term “character” in `NSString` method names refers to 16-bit platform-endian UTF-16 units. This is a common convention for string objects. In most cases, clients don't need to be overly concerned with this; as long as you are dealing with substrings, the precise interpretation of the range indexes is not necessarily significant.

The vast majority of Unicode code points used for writing living languages are represented by single UTF-16 units. However, some less common Unicode code points are represented in UTF-16 by surrogate pairs. A surrogate pair is a sequence of two UTF-16 units, taken from specific reserved ranges, that together represent a single Unicode code point. CFString has functions for converting between surrogate pairs and the UTF-32 representation of the corresponding Unicode code point. When dealing with `NSString` objects, one constraint is that substring boundaries usually should not separate the two halves of a surrogate pair. This is generally automatic for ranges returned from most Cocoa methods, but if you are constructing substring ranges yourself you should keep this in mind. However, this is not the only constraint you should consider.

In many writing systems, a single character may be composed of a base letter plus an accent or other decoration. The number of possible letters and accents precludes Unicode from representing each combination as a single code point, so in general such combinations are represented by a base character followed by one or more combining marks. For compatibility reasons, Unicode does have single code points for a number of the most common combinations; these are referred to as precomposed forms, and Unicode normalization transformations can be used to convert between precomposed and decomposed representations. However, even if a string is fully precomposed, there are still many combinations that must be represented using a base character and combining marks. For most text processing, substring ranges should be arranged so that their boundaries do not separate a base character from its associated combining marks.

In addition, there are writing systems in which characters represent a combination of parts that are more complicated than accent marks. In Korean, for example, a single Hangul syllable can be composed of two or three subparts known as jamo. In the Indic and Indic-influenced writing systems common throughout South and Southeast Asia, single written characters often represent combinations of consonants, vowels, and marks such as viramas, and the Unicode representations of these writing systems often use code points for these individual parts, so that a single character may be composed of multiple code points. For most text processing, substring ranges should also be arranged so that their boundaries do not separate the jamo in a single Hangul syllable, or the components of an Indic consonant cluster.

In general, these combinations—surrogate pairs, base characters plus combining marks, Hangul jamo, and Indic consonant clusters—are referred to as grapheme clusters. In order to take them into account, you can use `NSString`’s `rangeOfComposedCharacterSequencesForRange:` or `rangeOfComposedCharacterSequenceAtIndex:` methods, or `CFStringGetRangeOfComposedCharactersAtIndex`. These can be used to adjust string indexes or substring ranges so that they fall on grapheme cluster boundaries, taking into account all of the constraints mentioned above. These methods should be the default choice for programmatically determining the boundaries of user-perceived characters.:

In some cases, Unicode algorithms deal with multiple characters in ways that go beyond even grapheme cluster boundaries. Unicode casing algorithms may convert a single character into multiple characters when going from lowercase to uppercase; for example, the standard uppercase equivalent of the German character “ß” is the two-letter sequence “SS”. Localized collation algorithms in many languages consider multiple-character sequences as single units; for example, the sequence “ch” is treated as a single letter for sorting purposes in some European languages. In order to deal properly

with cases like these, it is important to use standard `NSString` methods for such operations as casing, sorting, and searching, and to use them on the entire string to which they are to apply. Use `NSString` methods such as `lowercaseString`, `uppercaseString`, `capitalizedString`, `compare:` and its variants, `rangeOfString:` and its variants, and `rangeOfCharacterFromSet:` and its variants, or their CFString equivalents. These all take into account the complexities of Unicode string processing, and the searching and sorting methods in particular have many options to control the types of equivalences they are to recognize.

In some less common cases, it may be necessary to tailor the definition of grapheme clusters to a particular need. The issues involved in determining and tailoring grapheme cluster boundaries are covered in detail in Unicode Standard Annex #29, which gives a number of examples and some algorithms. The Unicode standard in general is the best source for information about Unicode algorithms and the considerations involved in processing Unicode strings.

If you are interested in grapheme cluster boundaries from the point of view of cursor movement and insertion point positioning, and you are using the Cocoa text system, you should know that on OS X v10.5 and later, `NSLayoutManager` has API support for determining insertion point positions within a line of text as it is laid out. Note that insertion point boundaries are not identical to glyph boundaries; a ligature glyph in some cases, such as an “fi” ligature in Latin script, may require an internal insertion point on a user-perceived character boundary. See *Cocoa Text Architecture Guide* for more information.
