Doc. No: J16/98-0009
WG21/N1152
Date: 6 March, 1998
Project: PL C++
Reply to: Fred J. Tydeman
tydeman@tybor.com

PROBLEM WITH UCN MODEL IN C++

This is not an official statement from WG14 / J11 (C language), but I believe that both C and C++ need to work on this to get it right.

This is an extract of a public comment on the C9X CD. Many of these same points were raised by others at the last C9X meeting. Many of us now believe (after being show examples) that the UCN model in C++ (and C9X) is broken.

Item 2. UCN basic model

Category: Normative change to existing feature retaining the original intent
Committee Draft subsection: 5.1.1.2, 5.2.1, 6.1.2, Annex I
Title: Universal Character Names (UCNs): the basic model is wrong
Detailed description:

Background

Draft C9X proposes a new notation (e.g. \u098a or \U0000098A) for including arbitrary ISO 10646 (Unicode) characters in identifiers, strings, and comments. They work as follows:

* In translation phase 1, each multibyte source file character is replaced by the corresponding UCN. This occurs very early, even before trigraph replacements.

* In translation phase 5 (just after preprocessing), UCNs in char constants and string literals are converted to the execution character set.

* Some UCNs are valid in identifiers, but not others. E.g. \u098a (BENGALI LETTER UU) is valid, but \u2110 (SCRIPT CAPITAL I) is not valid.

Problems

Here are some problems with UCNs as they appear in the current draft.

* The current draft assumes that source file characters can be transliterated to Unicode and back again without loss of information. This assumption is incorrect and unrealistic. For example, ISO-2022-JP <ftp://ftp.isi.edu/in-notes/rfc1468.txt> cannot be converted to Unicode without losing information.

  As a trivial example, ISO-2022-JP distinguishes between ‘ESC (B’ (which switches to ASCII) and ‘ESC (J’ (which switches to JIS-Roman); but Unicode discards the distinction between ASCII and JIS-Roman for most characters, as it unifies most of the characters in the two sets. This means that the C9X draft effectively disallows the correct handling of ISO-2022-JP in string literals; an ISO-2022-JP string that contains ‘ESC (J’ is not guaranteed to contain the same ‘ESC (J’ after being translated to Unicode and back again.

  I believe that EBCDIC has a similar problem, as there are multiple representations of ‘[’ in EBCDIC but only one in Unicode.

* The use of \u collides with common usage in include directives.
  For example, ‘#include "h\ufeed.h"’ must be treated
  equivalently to ‘#include "h@.h", where @ is Unicode character
  FEED (ARABIC LETTER WAW ISOLATED FORM). This is not what the
  programmer likely intended, and C9X must not require this weird
  sort of case−folding.

* The current draft requires that the implementation maintain
  translation tables from the source and execution character set
  to Unicode, in order to properly stringize strings. This
  requirement is unrealistic. Many environments are not using
  Unicode yet, and do not have such tables. This problem is
  particularly acute for portable compilers like GCC, as there
  is no portable standard for accessing such transliteration tables.

* The current draft silently changes behavior when stringizing strings
  that contain multibyte characters. For example, if @ represents
  the Unicode character with code FEED (ARABIC LETTER WAW ISOLATED FORM),
  then

  ```
  #define str(s) #s
  printf("\t# of <%s> is <%s>\n", "@", str("@"));
  ```

  outputs something like this:

  ```
  # of <@> is <"\ufeed">
  ```

  whereas with C89 the program would output

  ```
  # of <@> is <"@">
  ```

  Clearly the C89 behavior is what is expected.

* The current draft precludes a simple implementation that simply
  treats characters with the top bit on as letters. Such an
  implementation supports popular encodings like UTF−8 and EUC
  without having to know what the encoding is. However, the
  current draft requires that the implementation convert each
  character to Unicode, which means the implementation must
  laboriously check for encoding errors, transliterate to and
  from Unicode, and the like. It also means the user must
  configure the compiler correctly, to specify which encoding is
  desired.

* The list of UCNs allowed in identifiers is arbitrary and weird.
  Although the standard seems to allow an implementation to permit
  almost all UCNs in identifiers, this is not made clear.