

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.4.3 Universal character names

universal character name syntax

universal-character-name:

`\u hex-quad`

`\U hex-quad hex-quad`

hex-quad:

hexadecimal-digit hexadecimal-digit

hexadecimal-digit hexadecimal-digit

Commentary

It is intended that this syntax notation not be visible to the developer, when reading or writing the source code that contains instances of this construct. A *universal-character-name* aware editor being used to display the ISO 10646 character represented by the numeric value specified by the *hex-quad* sequence value. Without such editor support, the whole rationale for adding these characters to C, allowing developers to read and write identifiers in their own language, is voided.

C90

Support for this syntactic category is new in C99.

Other Languages

Java calls this lexical construct a *UnicodeInputCharacter* (and does not support the `\U` form, only the `\u` one).

Coding Guidelines

It is difficult to imagine developers regularly using UCNs with an editor that does not display UCNs in some graphical form. A guideline recommending the use of such an editor would not be telling developers anything they did not already know.

A number of theories about how people recognize words have been proposed. One of the major issues yet to be resolved is the extent to which readers make use of *whole word* recognition versus mapping character sequences to sound (phonological coding). Support for UCNs increases the possibility that developers will encounter unfamiliar characters in source code. The issue of developer performance in handling unfamiliar characters is discussed elsewhere.

Example

```

1  #define foo(x)
2
3  void f(void)
4  {
5  foo("\u0123") /* Does not contain a UCN. */
6  foo(\u0123); /* Does contain a UCN. */
7  }
```

Constraints

A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 (\$), 0040 (@), or 0060 (‘), nor one in the range D800 through DFFF inclusive.⁶²⁾

Commentary

The ISO 10646 Standard defines the ranges 00 through 01F, and 07F through 09F, as the 8-bit control codes (what it calls *C0* and *C1*). Most of the UCNs with values less than 00A0 represent characters in the basic source character set. The exceptions listed enumerate characters that are in the Ascii character set, but not

UCNs not basic character set

ISO 10646

Word recognition models of

reading characters unknown to reader

in the basic source character set. The ranges 0D800 through DBFF and 0DC00 through 0DFFF are known as the surrogate ranges. The purpose of these ranges is to allow representation of rare characters in future versions of the Unicode standard.

This constraint means that source files cannot contain the UCN equivalent for any members of the basic source character set.

UCNs are not permitted to designate characters from the basic source character set in order to permit fast compilation times for C programs. For some real world programs, compilers spend a significant amount of time merely scanning for the characters that end a quoted string, or end a comment, or end some other token. Although, it is trivial for such loops in a compiler to be able to recognize UCNs, this can result in a surprising amount of overhead. Rationale

A UCN is constrained not to specify a character short identifier in the range 0000 through 0020 or 007F through 009F inclusive for the same reason: this avoids allowing a UCN to designate the newline character. Since different implementations use different control characters or sequences of control characters to represent newline, UCNs are prohibited from representing any control character.

C++

If the hexadecimal value for a universal character name is less than 0x20 or in the range 0x7F–0x9F (inclusive), or if the universal character name designates a character in the basic source character set, then the program is ill-formed. 2.2p2

The range of hexadecimal values that are not permitted in C++ is a subset of those that are not permitted in C. This means that source which has been accepted by a conforming C translator will also be accepted by a conforming C++ translator, but not the other way around.

Other Languages

Java has no such restrictions on the hexadecimal values.

Common Implementations

Support for UCNs is new in C99. It remains to be seen whether translator vendors decide to support any UCN hexadecimal value as an extension.

Example

```
1  \u0069\u006E\u0074 glob; /* Constraint violation. */
```

Description

817 Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Commentary

UCNs may also appear in comments. However, comments do not have a lexical structure to them. Inside a comment character, sequences starting with `\u` are not treated as UCNs by a translator, although other tools may choose to do so, in this context. The mapping of UCNs in character constants and string literals to the execution character set occurs in translation phase 5.

The constraint on the range of values that a UCN may take prevents them from being used to represent keywords.

⁸¹⁶ UCNs
not basic character set

C++

The C++ Standard also supports the use of universal character names in these contexts, but does not say in words what it specifies in the syntax (although 2.2p2 comes close for identifiers).

Other Languages

In Java, *UnicodeInputCharacters* can represent any character and is mapped in lexical translation step 1. It is possible for every character in the source to appear in this form. The mapping only occurs once, so `\u005cu005a` becomes `\u005a`, not `Z` (005c is the Unicode value for `\` and 005a is the Unicode character for `Z`).

Coding Guidelines

UCNs in character constants and string literals are used to represent characters that are output when a program is executed, or in identifiers to provide more readable source code. In the former case it is possible that UCNs from different natural languages will need to be represented. In the latter case it might be surprising if source code contained UCNs from different languages. This usage is a complex one involving issues outside of these coding guidelines (e.g., configuration management and customer requirements) and your author has insufficient experience to know whether any guideline recommendations might be worthwhile.

Some of the coding guideline issues relating to the use of characters outside of the basic execution character set are discussed elsewhere.

Example

```

1  #include <wchar.h>
2
3  int \u0386\u0401;
4  wchar_t *hello = "\u05B0\u0901";

```

multibyte
character
source contain

Semantics

short identifier The universal character name `\Unnnnnnnn` designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*.⁶³⁾ 818

Commentary

The standard specifies how UCNs are represented in source code. A development environment may chose to provide, to developers, a visible representation of the UCN that matches the glyph with the corresponding numeric value in ISO 10646. The ISO 10646 BNF syntax for short identifiers is:

```
{ U | u } [ {+}(xxxx | xxxxx | xxxxxx) | {-}xxxxxxxx ]
```

where *x* represents a hexadecimal digit.

Other Languages

Java does not support eight-digit universal character names.

Coding Guidelines

This form of UCN counts toward a greater number of significant characters in identifiers with external linkage and therefore is not the preferred representation. However, the developer may not have any control over the method used by an editor to represent UCNs. Given that characters from the majority of human languages can be represented using four-digit short identifiers, eight-digit short identifiers are not likely to be needed. If the development environment offers a choice of representations, use of four-digit short identifiers is likely to result in more significant characters being retained in identifiers having external linkage.

Similarly, the universal character name `\unnnn` designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is 0000nnnn). 819

ISO 10646
short identifier

external
identifier
significant
characters

Commentary

It was possible to represent all of the characters specified by versions 1 and 2 of the Unicode-sponsored character set using four-digit short identifiers. Version 3 introduced characters whose representation value requires more than four digits.

Other Languages

Java only supports this form of four-digit universal character names.

820 62) The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF-16).

footnote
62**Commentary**

Requiring that characters in the basic character set not be represented using UCN notation helps guarantee that existing tools (e.g., editors) continue to be able to process source files. [basic character set](#)

The control characters may have special meaning for some tools that process source files (e.g., a communications program used for sending source down a serial link).

C++

The C++ Standard does not make this observation.

821 63) Short identifiers for characters were first specified in ISO/IEC 10646–1/AMD9:1997.

footnote
63**Commentary**

This amendment appeared eight years after the first publication of the C Standard (which was made by ANSI in 1989).

References