ISO/IEC JTC1 SC22 WG14 N962
2001-11-05

Proposal for a work item

Additional character types

Summary

The aim of this work item is to introduce additional character data types and corresponding character and string literals. This will allow support for encodings such as UTF-16 in a portable way.

Reasons for supporting additional character data types

The programming language C as specified by the International Standard ISO/IEC 9899:1999 provides two data types for character processing: char and wchar_t. For these data types a set of library functions is also specified.

Data processing systems should nowadays be able to support many natural languages. This requires character sets that comprise more characters than can be represented in a single byte. Common approaches are to use a variable number of bytes for each character or to base the character encoding on 16-bit or 32-bit units.

The C Standard requires that the type wchar_t is capable of representing any character in the current locale. However, a minimum width is not guaranteed. On the other hand, the width of wchar_t may be large, implying excessive memory requirements for application programs. In multi-user environments as well as on mobile devices, memory is limited. Additional memory overhead means that cache limits will be exceeded more often and paging will occur more frequently.

Using wchar_t, the portability of programs is limited since the width is platform-dependent.

Some other programming languages support encodings based on 16-bit units or even use them as the processing character set. A number of database systems and some other commercially important software products offer APIs for UTF-16. (See the Appendix for details on UTF-16 and other Unicode encoding forms.) UTF-16 is also a viable choice when migrating from UCS-2 implementations.

When UTF-8 is used, byte-oriented APIs are sufficient, but for almost every commonly used language, the complexity of variable width characters shows up, except for the characters in the 7-Bit ASCII range.

Issues to be worked on

C provides integer data types that can, in principle, be used for characters. However, large applications contain numerous literal strings, even after all the user-visible strings have been externalised for internationalisation. Lack of support for literal strings is therefore a significant barrier to porting existing programs. It would be cumbersome to provide conversions at runtime to convert the literals from a multibyte encoding to the process encoding.

In this work, additional character data types will be defined. For these types, string literals and character literals need to be introduced. The encoding can either be made dependent on the active locale at compilation time or be specified as a specific Unicode encoding form. It has to be discussed whether an even higher degree of generality is desirable. For example, it is possible to design a mechanism that encompasses UTF-8 literals in an EBCDIC environment.

The C Standard already has the concept of Unicode-based universal character names, but this does not necessarily imply that Unicode has to become the preferred character set.

This work does not seek to change or invalidate the support for character sets currently used in C and C++ programs, but to make sure that portable support for a range of additional encodings is possible, including UTF-16.

In C++ it is possible to implement strings as parameterised classes. The parameter is the integer data type, which is used for the character strings. However there is currently no possibility to use literals of these types.

Functions that support the additional character data types are not the primary interest of this work. It has to be determined whether a (possibly minimal) set of library functions will be specified for the new character data types. For certain languages, and in particular for the UTF-16 encoding, it does not seem to be appropriate to use the model of processing one fixed-width code unit at a time. For this and other reasons it may not be favourable to carry the existing APIs for char and wchar_t strings over to any new character data type. A model that is oriented on variable-length strings would be preferable. Corresponding functions can be implemented using the current capabilities of the C (and C++) language.

History

In May 2001, the Unicode Technical Committee decided to suggest that JTC1 SC22/WG14 consider the support of a portable data type in the C/C++ language, which is based on UTF-16.

Appendix: Unicode Encoding Forms

(Partially based on the Unicode FAQ Copyright ©1998-2000 Unicode, Inc.)

For Unicode, there are three main encoding forms (UTF-8, UTF-16, and UTF-32), which encode the same common character repertoire. Transformations between these encoding forms are efficient and do not lead to any loss of data.

UTF-16 allows access to nearly 63500 characters as single Unicode 16-bit units. It can access one million additional characters by a mechanism known as surrogate pairs. Two ranges of Unicode code values are reserved for the high (first) and low (second) values of these pairs. Highs are from 0xD800 to 0xDBFF, and lows from 0xDC00 to 0xDFFF. Since the most common characters have already been encoded as 16-bit values, the characters requiring surrogate pairs will be relatively rare.

All characters represented in UTF-16, including those represented with a surrogate pair, can be represented as a single 32-bit unit in UTF-32. This single unit corresponds to the Unicode scalar value, which is the abstract number associated with a Unicode character. UTF-32 is a subset of the encoding called UCS-4 in ISO 10646.

In UTF-8, the representation of each character requires between one and four 8-bit units. For 7-bit ASCII characters, the UTF-8 encoding coincides with the ASCII encoding.

The UCS-2 encoding is a subset of UTF-16; it uses 16 bits for each character.