Doc. no.: N2035=06-0105
Date: 2006-05-23
Project: Programming Language C++
Subgroup: Library
Reply to: Matthew Austern <austern@google.com>

Minimal Unicode support for the standard library

Background

Unicode is an industry standard developed by the Unicode Consortium, with the goal of encoding every character in every writing systems. It is synchronized with ISO 10646, which contains the same characters and the same character codes, and for the purposes of this paper we may treat Unicode and ISO 10646 as synonymous. Many programming languages and platforms already support Unicode, and many standards, such as XML, are defined in terms of Unicode. There has already been some work to add Unicode support to ISO C++.

C++ has two character type, char and wchar_t. The standard does not specify which character set either type uses, except that each is a superset of the 95-character *basic execution character set*. In practice char is almost always an 8-bit type, typically used either for the ASCII character set or for some 256-character superset of ASCII (e.g. ISO-8859-1). Some programs use wchar_t for Unicode characters, but wchar_t varies enough from one platform to another that it is unsuitable for portable Unicode programming.

Unicode assigns "a unique number for every character, no matter what the platform, no matter what the program, no matter what the language." These numbers are known as *code points*. An *encoding scheme* specifies the way in which a sequence of code points is represented in an actual program. Code points rang from 0x000000 through 0x10ffff, so 21 bits suffice to represent all Unicode characters. No popular architecture has a 21-bit word size, so instead most programs that work with Unicode use one of the following encoding schemes for internal processing:

- UTF-32 uses a 32-bit word to store each character. This encoding is attractive because of its simplicity, unattractive because it wastes 11 bits per character.
- UTF-16 is a variable width encoding scheme where a code point is represented by either one or two 16-bit *code units*. The most common characters are represented as a single code unit, and the less common characters are represented as two code units, called *surrogates*. It is possible to tell, without having to examine context, whether a UTF-16 code unit is a leading surrogate, a trailing surrogate, or a complete character.
- UTF-8 is a variable-width encoding scheme where a code point is represented by one or more 8-bit code units.

Other encodings are sometimes used for serialization or external storage.

Unicode support for ISO C is described in TR 19769:2004, a Type 2 technical report. TR 19769 proposes two new character types, char16_t and char32_t, together with new syntax for character and string literals of those types, and a few additions to the C library to manipulate strings of those types. Lawrence Crowl's paper N2018, "Extensions for the Programming Language C++ to Support New Character Data Types," proposes that WG21 adopt TR 19769 almost unchanged; essentially the only change from TR 19769 is that char16_t and char32_t are required to be distinct from other integer types, so that it's possible to overload on them.

This paper describes changes to the standard library that will be needed if WG21 chooses to adopt N2018. It is a proposal for C++0x, because it proposes changes in existing standard library components.

Goals and design decisions

The main goal of this paper is simple: make it possible to use library facilities in combination with the two new character types char16_t and char32_t. This paper does not attempt to define new library facilities or to fix defects in existing ones, but only to make it possible to use char16_t and char32_t with existing library facilities.

This goal is important despite the existence of wchar_t. Even if wchar_t is the same size as one of those two types, it is distinct from both from the point of view of the C++ type system. It would be very poor user experience if we told users that they had to cast their Unicode strings to some other type in order to use library facilities, especially since that type would vary from one system to another. (Internally, of course, I imagine most library implementers will choose to share code between char32_t and wchar_t or between char16_t and wchar_t.) It is indeed irritating to have three distinct types when two of them will almost always be identical, but, as with char, signed char, and unsigned char, history leaves us little choice.

Minimal support for char32_t

Minimal support for char32_t is simple: UTF-32 is a fixed width encoding, so we just need to require specializations of all library facilities for char32_t in the same way that we do for char and wchar_t. Arguably a basic_string of 32-bit characters isn't all that useful, but I think just enough people would use it to make it worth having.

Minimal support for char16_t

Minimal support for char16_t is more complicated in theory, but equally simple in practice: again, just add specializations of all library facilities for char16_t. UTF-16 is not a fixed width encoding, but, for two reasons, it can almost be treated as one. First, UTF-16 is a fixed width encoding for text that contains only the most common 63K characters, and that’s most text. Second, since it’s always possible to tell whether a code unit is a complete character, a leading surrogate, or a trailing surrogate, there is little danger from treating a UTF-16 string as a sequence of code units instead of a sequence of code points. Corrupting a UTF-16 string by inserting an incorrect code unit is no more likely than corrupting a UTF-32 string, and the corruption, if any, will be confined to a single character.

We don’t need to say very much about how the library handles char16_t strings. There is already language in the standard to allow the ctype and codecvt facets to give errors at run time for invalid strings, and we need that for UTF-32 as well as UTF-16.

In practice, we need library support for UTF-16 because that’s the real world; if the standard library ignores UTF-16 then the standard library will be irrelevant to processing non-ASCII text. The small amount of extra simplicity that you get from using UTF-32 instead of UTF-16 just doesn’t outweigh the cost of using 4 bytes per character instead of 2+ε. Microsoft, Apple, and Java all use UTF-16 as their primary string representation, and in practice it works fine. Microsoft’s decision to use UTF-16 for wchar_t shows that there is no insuperable obstacle to using UTF-16 with the standard C++ library.

Names of template specializations

The C++ standard assigns names for many of the specializations of class templates on character types. For example, string is shorthand for basic_string<char> and wstreambuf is shorthand for basic_streambuf<wchar_t>. Our general pattern: no prefix for char specializations and the prefix ‘w’ for wchar_t specializations. What should the pattern be for specializations on char16_t and char32_t?

In principle we could use a prefix based on the “u” and “U” prefixes that N2018 proposes for Unicode string literals, or we could use a prefix or suffix based on the “16” and “32” in the type names

themselves. I propose a combination of the two: a "u" prefix for the char16_t specializations and a "u32" prefix for the char32_t specializations. Rationale:

- I expect that basic_string<char16_t> will be used much more often than basic_string<char32_t> or basic_string<wchar_t>, since UTF-16 is generally a good tradeoff between convenience and space efficiency. This argues that the name of basic_string<char16_t> should not be more cumbersome than that of basic_string<char32_t> or basic_string<wchar_t>, and that it should have a single-character prefix. The obvious choice is "u".

- There is an obvious one-character prefix for char32_t specializations, "U". In general, however, the standard library avoids uppercase names, and especially avoids having two names that differ only by case. Using a "u32" prefix for char32_t specializations does mean that it will be less convenient to use basic_string<char32_t> than to use basic_string<char16_t>, but that reflects what I expect to be real-world usage. I do not expect people to use UTF-32 as often as they use UTF-16.

Prior art

There is no prior art for a C++ library implementation containing four types named char, wchar_t, char16_t, and char32_t. However, there is also no doubt that the proposal is implementable. There is extensive prior art for C++ standard library implementations that use UTF-16 wide characters (wchar_t in Microsoft's C++ implementation uses UTF-16), and there is extensive prior art for C++ standard library implementations that use UTF-32 wide characters (most Unix implementations).

Dependence on N2018

Since N2018 has not been voted into the WP, building library facilities on top of it is slightly dicey. In principle a number of questions are still open: the names of the two new types (I have chosen to use char16_t and char32_t in accordance with TR 19769 and N2018), whether they are new built-in types or new user-defined types (the latter would only make sense if there are core language changes to permit string literals for user-defined types), whether char16_t and char32_t are the names of types or just the names of typedefs for underlying types with uglier names, and, if they are typedef names, which namespace the typedefs live in. Except for the names char16_t and char32_t, nothing in this paper depends on those decisions.

Possible future directions

Two items are conspicuously missing from this paper: UTF-8 support, and explicit support for Unicode features like normalization, case conversion, and collation. I intend to address those issues in future papers.

UTF-8 support

One way to provide UTF-8 support is in the form of a new string class whose interface is very different from basic_string. The reason it would have to be very different is that UTF-8 is much more fragile than UTF-16. The basic_string interface allows clients to update individual bytes, which is both useless and dangerous for UTF-8 strings. It would be better to have an interface that preserves string validity and that encourages users to view the string as code points instead of individual bytes. The interface of this class would support iostreams, provide access to individual 32-bit code points through read-only forward iterators, and provide read-only access to a char* that points to the underlying bytes in a UTF-8 encoding. Mutation of this class would be highly restricted.

The main argument in favor of a UTF-8 string class is Bjarne Stroustrup's observation that new

language features ought to come with new keywords. His reasoning applies to library features as well core language features. Since UTF-8 support is a new feature, there should be a marker to indicate the presence of that new feature. A class alerts users to the fact that the library supports UTF-8, and it alerts programmer X, reading programmer Y's code, that a sequence of bytes should be interpreted as a variable-width UTF-8 string rather than a string where each byte represents a single character.

The real question isn't whether we should support UTF-8 in the form of a new string class, but whether we need to support UTF-8 at all. There's a case to be made that we shouldn't. Some people believe that UTF-8 should only be used as an external representation and that internal processing should always be UTF-16 or UTF-32. My counterargument: that's one valid programming style, but this committee doesn’t need to choose between styles. There's an awful lot of real-world code that uses UTF-8 even internally, and programmers need UTF-8 to interface with third-party libraries like libxml2.

Unicode text manipulation

Unicode is more than a character set and a handful of encoding schemes. It also specifies a great deal of information about each character, including script identification, character classification, and text direction, and various operations on strings, including normalization, case conversion, and collation.

Normalization is particularly important because there are cases where two different sequences of code points can represent what is conceptually the same string. For example, a string that is printed as “á” can be either the single character U+00E1 (LATIN SMALL LETTER A WITH ACUTE) or the two-character sequence U+0061 (LATIN SMALL LETTER A) U+0301 (COMBINING ACUTE ACCENT). Unicode defines several different canonical forms, and algorithms for converting to canonical form and for testing string equivalence.

In principle some of these facilities are already part of C++’s facet interface, and it might be argued that we do not need a separate mechanism just for Unicode. There is, however, an important way in which Unicode is special: since it uses a single code point space for all scripts, many operations in Unicode are locale-independent that in other encodings are necessarily locale-dependent. Since the C++ locale interface is so awkward, it would be useful to provide a locale-independent interface for common operations that do not require locales.

ICU (International Components for Unicode) is a useful source of prior art.

Proposed working paper changes

Add two new sections after 21.1.3.2 [lib.char.traits.specializations.wchar.t]:

[lib.char.traits.specializations.char16.t]

```
namespace std {
  template<>
  struct char_traits<char16_t> {
    typedef char16_t char_type;
    typedef uint_least_16_t int_type;
    typedef streamoff off_type;
    typedef ustreampos pos_type;
    typedef mbstate_t state_type;
    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
    const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
```

```
    static char_type* assign(char_type* s, size_t n, char_type a);
    static int_type not_eof(const int_type& c);
    static char_type to_char_type(const int_type& c);
    static int_type to_int_type(const char_type& c);
    static bool eq_int_type(const int_type& c1, const int_type& c2);
    static int_type eof();
  };
}
```

The header <string> (21.2) declares a specialization of the class template char_traits for char16_t.

The two-argument member assign is defined identically to the built-in operator =. The two-argument members eq and lt are defined identically to the built-in operators == and <.

The member eof() returns an implementation defined constant that cannot appear as a valid UTF-16 code unit.

[lib.char.traits.specializations.char32.t]

```
namespace std {
  template<>
  struct char_traits<char32_t> {
    typedef char32_t char_type;
    typedef uint_least_32_t int_type;
    typedef streamoff off_type;
    typedef u32streampos pos_type;
    typedef mbstate_t state_type;
    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
    const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static int_type not_eof(const int_type& c);
    static char_type to_char_type(const int_type& c);
    static int_type to_int_type(const char_type& c);
    static bool eq_int_type(const int_type& c1, const int_type& c2);
    static int_type eof();
  };
}
```

The header <string> (21.2) declares a specialization of the class template char_traits for char32_t.

The two-argument member assign is defined identically to the built-in operator =. The two-argument members eq and lt are defined identically to the built-in operators == and <.

The member eof() returns an implementation defined constant that does not represent a Unicode code point.

In clause 21.2 [lib.string.classes], add the following to the beginning of the header <string> synopsis:

```
template<> struct char_traits<char16_t>;
template<> struct char_traits<char13_t>;
```

and the following to the end:

```
typedef basic_string<char16_t> ustring;
typedef basic_string<char32_t> u32string;
```

In Table 65 (Locale category facets) in clause 22.1 [lib.locale.category], add the following specializations:

```
collate<char16_t>
collate<char32_t>

ctype<char16_t>
ctype<char32_t>
codecvt<char16_t, char, mbstate_t>
codecvt<char32_t, char, mbstate_t>

moneypunct<char16_t>
moneypunct<char16_t, true>
moneypunct<char32_t>
moneypunct<char32_t, true>
money_get<char16_t>
money_get<char32_t>
money_put<char16_t>
money_put<char32_t>

numpunct<char16_t>
numpunct<char32_t>
num_get<char16_t>
num_get_char32_t>
num_put<char16_t>
num_put<char32_t>

time_get<char16_t>
time_get<char32_t>
time_put<char16_t>
time_put<char32_t>

messages<char16_t>
messages<char32_t>
```

In Table 66 (Required Specializations) in clause 22.1 [lib.locale.category], add the following specializations:

```
collate_byname<char16_t>
collate_byname<char32_t>

ctype_byname<char16_t>
ctype_byname<char32_t>
codecvt_byname<char16_t, char, mbstate_t>
codecvt_byname<char32_t, char, mbstate_t>

moneypunct_byname<char16_t, International>
moneypunct<char32_t, International>

numpunct_byname<char16_t>
numpunct_byname<char32_t>

time_get<char16_t, InputIterator>
time_get_byname<char16_t, InputIterator>
time_get<char32_t, InputIterator>
time_get_byname<char32_t, InputIterator>

time_put<char16_t, OutputIterator>
time_put_byname<char16_t, OutputIterator>
time_put<char32_t, OutputIterator>
time_put_byname<char32_t, OutputIterator>

messages_byname<char16_t>
messages_byname<char32_t>
```

In clause 22.2.1.4 [lib.locale.codecvt] paragraph 3, remove the phrase “namely codecvt<wchar_t, char,

mbstate_t> and codecvt<char, char, mbstate_t>." Add the following sentence, after the one describing the wchar_t specialization: "The specialization codecvt<char16_t, char, mbstate_t> converts between the UTF-16 and UTF-8 encoding schemes, and the specialization codecvt<char32_t, char, mbstate_t> converts between the UTF-32 and UTF-8 encoding schemes."

In clause 27.2 [lib.iostream.forward] add the following declarations to the header <iosfwd> synopsis:

```
typedef basic_ios<char16_t> uios;
typedef basic_streambuf<char16_t> ustreambuf;
typedef basic_istream<char16_t> uistream;
typedef basic_ostream<char16_t> uostream;
typedef basic_iostream<char16_t> uiostream;
typedef basic_stringbuf<char16_t> ustringbuf;
typedef basic_istringstream<char16_t> uistringstream;
typedef basic_ostringstream<char16_t> uostringstream;
typedef basic_stringstream<char16_t> ustringstream;
typedef basic_filebuf<char16_t> ufilebuf;
typedef basic_ifstream<char16_t> uifstream;
typedef basic_ofstream<char16_t> uofstream;
typedef basic_fstream<char16_t> ufstream;
typedef fpos<char_traits<char16_t>::state_type> ustreampos;

typedef basic_ios<char32_t> u32ios;
typedef basic_streambuf<char32_t> u32streambuf;
typedef basic_istream<char32_t> u32istream;
typedef basic_ostream<char32_t> u32ostream;
typedef basic_iostream<char32_t> u32iostream;
typedef basic_stringbuf<char32_t> u32stringbuf;
typedef basic_istringstream<char32_t> u32istringstream;
typedef basic_ostringstream<char32_t> u32ostringstream;
typedef basic_stringstream<char32_t> u32stringstream;
typedef basic_filebuf<char32_t> u32filebuf;
typedef basic_ifstream<char32_t> u32ifstream;
typedef basic_ofstream<char32_t> u32ofstream;
typedef basic_fstream<char32_t> u32fstream;
typedef fpos<char_traits<char32_t>::state_type> u32streampos;
typedef fpos<char_traits<char32_t>::state_type> u32streampos;
```

In clause 27.2 [lib.iostream.forward] paragraph 7, replace "char or wchar_t" with "char, wchar_t, char16_t, or char32_t". Change paragraph 9 to read "The types streampos, wstreampos, ustreampos, and u32streampos are used for positioning streams specialized on char, wchar_t, char16_t, and char32_t, respectively."

In clause 27.3 [lib.iostream.objects], add the following declarations to the header <iostream> synopsis:

```
extern uistream ucin;
extern uostream ucout;
extern uostream ucerr;
extern uostream uclog;

extern u32istream u32cin;
extern u32ostream u32cout;
extern u32ostream u32cerr;
extern u32ostream u32clog;
```

Add the following two new sections after 27.3.2 [lib.wide.stream.objects]:

[lib.char16.stream.objects]

```
uistream ucin;
```

The object ucin controls input from a stream buffer associated with the object stdin, declared in <cstdio>.

After the object ucin is initialized, ucin.tie() returns &ucout. Its state is otherwise the same as

required for basic_ios<char16_t>::init.

`uostream ucout;`
The object ucout controls output to a stream buffer associated with the object stdout, declared in <cstdio>.

`uostream ucerr;`
The object ucerr controls output to a stream buffer associated with the object stderr, declared in <cstdio>.

After the object ucerr is initialized, ucerr.flags() & unitbuf is nonzero and ucerr.tie() returns ucout. Its state is otherwise the same as required for basic_ios<char16_t>::init.

`uostream uclog;`
The object uclog controls output to a stream buffer associated with the object stderr, declared in <cstdio>.

[lib.char32.stream.objects]
`u32istream u32cin;`
The object u32cin controls input from a stream buffer associated with the object stdin, declared in <cstdio>.

After the object u32cin is initialized, u32cin.tie() returns &u32cout. Its state is otherwise the same as required for basic_ios<char32_t>::init.

`u32ostream u32cout;`
The object ucout controls output to a stream buffer associated with the object stdout, declared in <cstdio>.

`u32ostream u32cerr;`
The object u32cerr controls output to a stream buffer associated with the object stderr, declared in <cstdio>.

After the object u32cerr is initialized, u32cerr.flags() & unitbuf is nonzero and u32cerr.tie() returns u32cout. Its state is otherwise the same as required for basic_ios<char32_t>::init.

`u32ostream u32clog;`
The object u32clog controls output to a stream buffer associated with the object stderr, declared in <cstdio>.

In clause 28.4 [lib.re.syn], add the following declarations:

```
//28.8, class template basic_regex
typedef basic_regex<char16_t> uregex;
typedef basic_regex<char32_t> u32regex;

// 28.9, class template sub_match
typedef sub_match<const char16_t*> ucsub_match;
typedef sub_match<const char32_t*> u32csub_match;
typedef sub_match<ustring::const_iterator>  ussub_match;
typedef sub_match<u32string::const_iterator> wssub_match;

//28.10, class template match_results:
typedef match_results<const char16_t*> ucmatch;
typedef match_results<const char32_t*> u32match;
typedef match_results<ustring::const_iterator> usmatch;
typedef match_results<u32string::const_iterator> u32smatch;

// 28.12.1, class template regex_iterator
typedef regex_iterator<const char16_t*> ucregex_iterator;
```

```
typedef regex_iterator<const char32_t*> u32cregex_iterator;
typedef regex_iterator<ustring::const_iterator> usregex_iterator;
typedef regex_iterator<u32string::const_iterator> u32sregex_iterator;

// 28.12.2, class template regex_token_iterator
typedef regex_token_iterator<const char16_t*> ucregex_token_iterator;
typedef regex_token_iterator<const char32_t*> u32cregex_token_iterator;
typedef regex_token_iterator<ustring::const_iterator> usregex_token_iterator;
typedef regex_token_iterator<u32string::const_iterator> u32sregex_token_iterator;
```

References

Lawrence Crowl, *Extensions for the Programming Language C++ to Support New Character Data Types.* WG21 N2018, 2006.

The Unicode Consortium. The Unicode Standard, Version 4.1.0, defined by: *The Unicode Standard, Version 4.0* (Boston, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1), as amended by Unicode 4.0.1 (http://www.unicode.org/versions/Unicode4.0.1) and by Unicode 4.1.0 (http://www.unicode.org/versions/Unicode4.1.0).

The Unicode Consortium, *Frequently Asked Questions*, http://www.unicode.org/unicode/faq/. See in particular http://www.unicode.org/faq/utf_bom.html for a discussion of encodings.

ISO. *Information technology -- Universal Multiple-Octet Coded Character Set (UCS)*, ISO/IEC 10646.

ISO. *Information technology -- Programming languages, their environments and system software inferfaces -- Extensions for the programming language C to support new character data types*, ISO/IEC TR 19769:2004.