;login:
THE MAGAZINE OF USENIX & SAGE
August 2002 volume 27 • number 4

inside:

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

wide characters

by Glen McCluskey

Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

glenm@glenmccl.com

We've been looking at some of the new features in C99, the standards update to C. In this column we'll consider features added to the language and library in support of wide characters, as typically used with foreign languages.

Character Sets

Several terms are used in the standard to describe C character sets. The first of these is "basic character set" and refers to a set of single-byte characters that all C99 implementations must support. Roughly speaking, this character set is 7-bit ASCII without some of the control characters. It consists of printable characters like A–Z and 0–9, along with tab, form feed, and so on.

The basic character set is divided into source and execution character sets, and these differ slightly. For example, the basic execution character set is required to have a null character (\0), used as a string terminator.

The extended character set is a superset of the basic character set, and adds additional locale-specific characters. It, too, is divided into source and execution character sets.

A wide character is a character of type wchar_t, and is capable of representing any character in the current locale. In other words, a wide character may be a character from either the basic character set, such as the letter A, or a character from the extended character set.

Wide Character Constants

Let's look at some actual examples of wide character usage. The first demo program prints the size of wchar_t on your local system:

```
#include <stdio.h>
#include <wchar.h>

int main()
{
    printf("sizeof(wchar_t) = %u\n", sizeof(wchar_t));
}
```

When I run this program on my Linux system, the result is:

```
sizeof(wchar_t) = 4
```

wchar_t is a signed or unsigned integral type big enough to hold all the characters in the local extended character set, and is a 32-bit long on my system.

wide character constants are specified similarly to normal character constants, with a preceding L before the constant:

```
#include <stdio.h>
#include <wchar.h>

int main()
{
    wchar_t wc1 = L'a';
    printf("%lx\n", wc1);

    wchar_t wc2 = L'\377';
    printf("%lx\n", wc2);

    wchar_t wc3 = L'\x12345678';
    printf("%lx\n", wc3);
}
```

When I run this program, the result is:

```
61
ff
12345678
```

In the first two cases, the wide character is stored in the least significant byte of the long, while in the last case, all four bytes of the long are used to represent a single wide character.

This example illustrates a confusing point about wide characters – the idea of multiple representations. For example, wc3 is initialized with a wide character constant, a constant that requires 13 bytes to express in the source program. The constant itself is stored in four bytes during execution (in a 32-bit long). And a little later on, we'll see examples of what is called "state-dependent encoding," a mechanism used to encode wide characters as a stream of bytes (1–6 bytes per wide character, on my system). This encoding is used for writing wide characters to a file.

The term "multibyte character" is defined to be a sequence of one or more bytes that represents a single character in the extended source or execution environment. A character from the extended character set can have several different representations. These representations may appear in source code, in the execution environment, or in data files.

Here's another example, showing how wide character strings are specified:

```
#include <assert.h>
#include <wchar.h>

int main()
{
    wchar_t* wstr1 = L"testing\x12345678";
    wchar_t wstr2[] = L"testing\x12345678";

    assert(*wstr1 == 't');
    assert(*(wstr1 + 7) == 0x12345678);

    assert(wstr2[0] == 't');
    assert(wstr2[7] == 0x12345678);
}
```

String Operations

Many familiar operations are supported on wide character strings. For example, here's a demo that implements a function to convert to lowercase:

```
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

wchar_t* tolower(wchar_t* str)
{
    wchar_t* start = str;

    // convert each wide character to lowercase

    for (; *str; str++) {
        if (iswupper(*str))
            *str = tolower(*str);
    }

    return start;
}

int main()
{
    wchar_t* str = L"TESTing";
    wchar_t buf[8];

    wcscpy(buf, str);

    tolower(buf);

    printf("%ls\n", buf);
}
```

Note that the definition of an uppercase character may be locale specific.

Wide Characters and I/O

Suppose that you have a wide character string and you'd like to write it to a file and then read it back. How can you do this? Here's one approach:

```
#include <assert.h>
#include <stdio.h>
#include <wchar.h>

int main()
{
    // write a wide character string to a file

    FILE* fp = fopen("outfile", "w");
    assert(fp);
    fwprintf(fp, L"string is\377: %ls\n", L"TESTing\x1234");
    fclose(fp);

    // read the characters of the string back from the
    // file

    fp = fopen("outfile", "r");
    assert(fp);
    wint_t c;
    wchar_t buf[25];
    size_t len = 0;
    while ((c = getwc(fp)) != WEOF)
        buf[len++] = c;
    fclose(fp);
    buf[len] = 0;

    // check results

    if (wcscmp(buf, L"string is\377: TESTing\x1234\n")
                == 0)
        printf("strings are equal\n");
    else
        printf("strings are unequal\n");
}
```

Much of this code is identical to what you would use when reading and writing regular strings of bytes.

wint_t is a type that is related to wchar_t in a way similar to the relationship between int and char; it can hold all possible wchar_t values, as well as one distinguished value that is not part of the extended character set (WEOF).

The only other tricky thing in this example is stream orientation, something that's implicit in the code. A file stream can be either byte or wide oriented. The orientation is determined by the first operation on the stream, or explicitly via the fwide() call. Since the first write operation in the demo is fwprintf(), and the first read operation is getwc(), and these are wide character functions, the streams are marked as having wide orientation.

Why does stream orientation matter? The reason is that an encoding may be applied to wide characters written to a file. Suppose you are programming with wide characters, you need to do wide character I/O, and your wide characters are four bytes long when using the wchar_t representation. One way of writing such characters to a file is to actually write four bytes for each character.

But what happens if, most of the time, the values of your wide characters are within the range of 7-bit ASCII? In such a case, three zero bytes will be written for each character. And the resulting files will not be readable by tools that expect ASCII. This problem exists today, for example, with tools that write 16-bit Unicode to a file. One solution to this problem is to encode characters such that 7-bit ASCII is represented as itself, that is, a single byte, while other character values are encoded using multiple bytes.

But if an encoding is applied, then it no longer makes sense to mix byte and wide-file operations. This is especially true given that an encoding may be state dependent, and dipping into a byte stream in the middle of a multiple-byte encoding of a wide character has no meaning.

Encodings

Let's look a little deeper into the encoding issue, with another example. This demo converts wide character values into sequences of encoded bytes:

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main()
{
      char buf[MB_CUR_MAX];
      int n;

      // convert a single-byte character to a multibyte
      // character

      n = wctomb(buf, L'a');
      printf("len = %d\n", n);
      for (int i = 0; i < n; i++)
          printf("%hhx ", buf[i]);
      printf("\n");

      // convert another single-byte character

      n = wctomb(buf, L'\377');
      printf("len = %d\n", n);
      for (int i = 0; i < n; i++)
          printf("%hhx ", buf[i]);
      printf("\n");

      // convert a wide character

      n = wctomb(buf, L'\x12345678');
      printf("len = %d\n", n);
      for (int i = 0; i < n; i++)
      printf("%hhx ", buf[i]);
      printf("\n");
}
```

The output is:

```
len = 1
61
len = 2
c3 bf
len = 6
fc 92 8d 85 99 b8
```

The character a is encoded as itself, while the character \377 is encoded as two bytes 0xc3 and 0xbf. The constant L'\x12345678', internally represented as a four-byte long, is encoded using six bytes.

Here's another example of encoding and decoding:

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main()
{
      char buf[MB_CUR_MAX];
      wchar_t wc1 = L'\x12345678';
      wchar_t wc2;
      int n, nn;

      // convert a wide character to a multibyte
      // character

      n = wctomb(buf, wc1);
      printf("%d\n", mblen(buf, n));

      // reverse the process

      nn = mbtowc(&wc2, buf, n);

      // check result

      if (wc1 == wc2 && n == nn)
          printf("equal\n");
      else
          printf("unequal\n");
}
```

The wctomb() function encodes a wide character into a stream of bytes, and mbtowc() reverses the process.

Restartable Functions

Consider the second part of the last example. The processing of the first part of the example – a wide character encoded into a buffer of one or more bytes – was reversed, by taking the buffer and converting it back into a wide character.

In a real-world example, things might not be quite as simple. For instance, you might have an application where bytes are coming in across a network one at a time, and several of the bytes put together represent a wide character. You'd somehow like to keep track of the state of the decoding as each byte

comes in, and when a valid wide character is detected, process it.

As part of support for wide characters, C99 has a set of what are called restartable functions. The idea is that you initialize a state object used to keep track of the encoding or decoding state, and then you pass this object to the functions. Let's see how this idea works in practice:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>

int main()
{
    // convert a wide character into a byte stream

    char buf[MB_CUR_MAX];
    wchar_t wc1 = L'\x12345678';
    int len = wctomb(buf, wc1);
    printf("len = %d\n", len);

    // initialize mbstate_t object

    wchar_t wc2;
    mbstate_t mbstate;
    memset(&mbstate, 0, sizeof(mbstate_t));
    size_t retval;

    // convert the first len - 1 bytes of the byte stream

    retval = mbrtowc(&wc2, buf, len - 1, &mbstate);
    printf("retval = %d\n", retval);

    // convert the last byte of the byte stream

    retval = mbrtowc(&wc2, &buf[len - 1], 1, &mbstate);
    printf("retval = %d\n", retval);

    // compare with original

    if (wc1 == wc2)
        printf("equal\n");
    else
        printf("unequal\n");
}
```

In the first part of the example, a wide character is encoded into a stream of bytes. We then initialize an mbstate_t object and convert the stream of bytes back to a wide character. But in the first call to mbrtowc(), we omit the last input byte, implying that the conversion cannot be completed during this function call. The state object captures an intermediate state of conversion. The state object is then passed to the second mbrtowc() call, and the conversion is completed.

The result of running the demo is:

```
len = 6
retval = -2
retval = 1
equal
```

The initial -2 return value from the first mbrtowc() call indicates that a valid partial encoded wide character was found in the input byte stream.

Wide character support is especially useful if you're working with foreign languages. C applications often assume English and ASCII are being used, and the wide character type and library functions add support for other character sets.