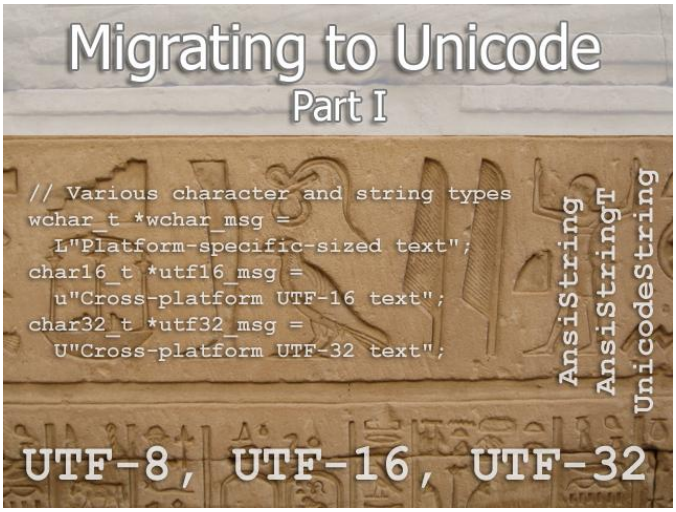
Migrating to Unicode, Part I

By Josh Kelley

Versions: C++Builder 2010, 2009



One of the biggest changes in C++Builder 2009 and 2010 is the addition of full Unicode support throughout the VCL and RTL. Unicode support is a major step forward for the VCL and is critical for internationalization, but its implementation as an absolute requirement within the VCL can seem like a major obstacle in upgrading to C++Builder 2009 or 2010.

Fortunately, migrating to Unicode can be much less daunting than it first appears. The first key realization in migrating to Unicode for C++Builder 2009 or 2010 is this: You do not have to migrate to Unicode to use C++Builder 2009 or 2010. C++ is a diverse language, permitting the use of many libraries and several programming paradigms, and while your code that uses the VCL needs to be Unicode-aware, your code that uses the C runtime library, the STL, or the Windows API (for example) can continue to use ANSI and only convert to Unicode when passing data to or from the VCL. A complete migration to Unicode is obviously necessary to gain the full benefits of internationalization, but migrating only the VCL portion of your code can drastically simplify the task of upgrading to C++Builder 2009 or 2010 while letting you gain the other significant benefits that those versions offer (such as Boost, C++0x support, gestures, and a Ribbon control).

Whether you choose to make your entire application Unicode-aware or to upgrade only the portions that interact with the VCL, there are several C and C++ development techniques that can make the task much easier. Part I of this article offers an introduction to Unicode and discusses working with Unicode in C, C++, and the VCL, while Part II examines some of these Unicode migration techniques in more detail.

Remember, though, that supporting Unicode is only part of internationalizing an application. Full internationalization also includes issues such as date and time format, sorting orders, support for right-to-left layout, culture-neutral icon design, and so on. Such issues are beyond the scope of this article.

An introduction to Unicode

This is a brief introduction to Unicode. For a more thorough background, see [1] or [2].

Unicode is the international standard for representing text from almost any language in a computer. Unicode was designed to replace the older ANSI and ASCII standards and to address those standards' disadvantages in dealing with international text.

ASCII (such as was used on the original IBM PCs) was designed for plain English text only. It represented each character as a number between 32 and 127. Space was 32, '0' was 48, 'A' was 65, and so on. ASCII characters were stored one per byte, but since ASCII only defined characters through 127, byte values 128-255 were assigned a number of different meanings depending on where they were used. (The original IBM PC assigned various accented characters and line drawing characters to the 128-255 range; later, various countries assigned letters from their own languages' alphabets; and so on.)

The ANSI standard kept characters 32-127 the same as ASCII but standardized the use of the 128-255 range into a series of *code pages*. Each language or region could be assigned its own code page, and as long as 8-bit textual data was associated with a code page, it could faithfully represent non-plain-English text. ("ANSI" is actually a bit of a misnomer; Windows' code pages were never standardized by ANSI.)

The Unicode standard was introduced to cover two major shortcomings with the ANSI standard. First, the ANSI standard made no provision for multilingual text that needed more than one code page. Second, alphabets such as Chinese have thousands of characters and so cannot fit in a single code page.

Unicode provides a standard way of referring to any of over 100,000 characters ([2]). Each of these 100,000 unique characters (called *code points* in Unicode terminology) is assigned a unique name and a unique numeric identifier, which is written as "U+" followed by a 4-digit hexadecimal value. For example, the code point for the English capital A is named "LATIN CAPITAL LETTER A" and is written U+0041. It is important to note that Unicode characters *are not guaranteed to be representable by a 16-bit value*. The portion of Unicode characters that *can* be represented in 16 bits is called the *Basic Multilingual Plane*. Characters outside of the Basic Multilingual Plane are primarily used for ancient scripts (such as Egyptian hieroglyphics), musical notation, and rarely used Han ideograms.

Because code points are abstract entities, Unicode provides several *encodings* to represent these abstract code point values in a machine-readable form. An encoding describes how to represent each *code point* as one or more *code units*. (A code unit is simply "the minimal bit combination that can represent a unit of encoded text" [3].) The most common encodings are UTF-8, UTF-16, and UTF-32.

- UTF-32 uses 32-bit code units. UTF-32 has the advantage that each code point takes a consistent amount of space, but it tends to be wasteful of space (since English characters can otherwise be represented in 8 bits, and most characters in use can otherwise be represented in 16 bits). Because of its high storage requirements, UTF-32 is rarely used.
- UTF-16 uses 16-bit code units. UTF-16 can represent *most* of the code points in use as a single code unit. Code points above U+FFFF must be represented using *surrogate pairs*, a pair of code units that together represent a single code point. UTF-16 has the advantage that *most* code points can be represented as single code units and thus take a consistent amount of space; however, the storage requirements are higher than UTF-8, and the possibility of surrogate pairs is easy to overlook in development and testing. UTF-16 is the preferred encoding for a number of platforms and libraries (including Windows, OS X, Java, and .NET).
- UTF-8 uses 8-bit code units. Byte values between 32 and 127 are identical to ASCII, and UTF-8 strings can use a terminating NULL character just like standard C ASCII strings. Code points above 127 are represented by sequences of up to 4 bytes. UTF-8 has the advantage of being backwards compatible with ASCII; it has the disadvantage that code units frequently take varying amounts of space (and so operations like "take the 100th character from this string" can become much harder). Because UTF-8 has lower space requirements, it is commonly used for web pages, email, and similar stored or transmitted data. It is also the preferred encoding for a few platforms and libraries (including the Linux kernel and the GTK framework).

An additional complication of Unicode is the existence of composite (or composable) characters. A letter with accent marks or diacritical marks can be represented in Unicode either as a unique, precomposed code point or as the code point for the plain letter followed by the code point for its diacritical mark (or possibly multiple code points for multiple diacritical marks). For example, é can be represented in Unicode either as U+00E9 ("Latin small letter e with acute") or as U+0065 ("Latin small letter e") followed by U+0301 ("combining acute").

The existence of composable characters means that a simple binary comparison is not sufficient for checking two Unicode strings for equality. Because of composable characters, UTF-16's surrogate pairs, and UTF-8's variable length encodings, it is no longer valid to assume that accessing arbitrary characters out of a string or splitting string at arbitrary indexes will work. Operating system APIs and third-party libraries such as ICU [4] offer routines to help with these complications. (Windows APIs in particular will be discussed below under "Unicode in the Windows API.")

Working with Unicode

Unlike languages which offer a single built-in string data type, C++Builder offers several choices: code can use C-style characters and strings, or C++ `string` objects, or VCL `String` objects. Each of these has its own

set of Unicode variations. Similarly, the Windows API provides both ANSI and Unicode variants.

Unicode data in C

A C-style string is simply an array of `char` values, terminated by a NULL byte (also written as '`\0`'). Each `char` takes one byte of storage. (This is guaranteed by the C and C++ standards; as a pedantic note, however, one byte of storage is not guaranteed to be 8 bits, and some rare platforms use 16 bits or other sizes.) The encoding of a `char` string is not specified; it could be straight ASCII, or any of the ANSI code pages, or even UTF-8, although in Windows, it's generally assumed to be in the system default ANSI code page. Working with Unicode introduces several more C and C++ data types for C-style strings:

- `wchar_t`: C and C++ apps have traditionally used `wchar_t` as a replacement for `char` when working with Unicode strings. `wchar_t` strings are written as `L"Hello, world! \u263A"`. The size of a `wchar_t` is *compiler-dependent*: on Windows, it's 16 bits and assumed to contain UTF-16 data; but on Linux, it's 32-bits; and other platforms may use values as small as 8 bits. If you need truly cross-platform Unicode-aware code, you may need to avoid the built-in types altogether and instead use a third-party library such as ICU [4].
- `char16_t`, `char32_t`: C++0x (the draft of the new standard for the C++ language) specifies these two new character types for holding UTF-16 and UTF-32 data, respectively. `char16_t` values are written as `u"Hello, world! \u263A"`. `char32_t` values are written as `U"Hello, world! \u263A"`. (C++0x also allows using `u8"Hello, world! \u263A"` to represent UTF-8 data as a `char` array, but C++Builder doesn't support this.)

Of course, having Unicode data types does little good if you have no way to specify some of the more esoteric Unicode characters. Traditional C strings can represent nonprintable characters using predefined escape characters like \n (newline) as well as arbitrary hexadecimal values like \x7f. Similarly, Unicode characters can be written as \u followed by their 4-digit hexadecimal value. (For example, `L"\u00E9"` is a "Latin small letter e with acute," and `L"\u263A"` is a smiley face.)

Because the C++Builder IDE is fully Unicode-aware, you can also directly enter Unicode characters into your code, without resorting to escape characters. There are a few ways to enter Unicode characters in Windows: for example, you can use Windows' built-in Character Map utility, or you can hold the Alt key while typing '+' followed by the Unicode character's 4-digit hexadecimal value. The fileformat.info web site has a complete list of input methods for Windows [5] as well as a searchable database of Unicode characters [6]. C++Builder automatically uses the UTF-8 encoding for source files containing Unicode characters.

Sprinkling Unicode escape characters throughout your code hampers readability, and directly entering Unicode characters can sometimes be difficult to work with. An alternative is to use preprocessor `#defines` to create macros for Unicode characters which your application needs. Using preprocessor macros instead of `const` values is often discouraged in modern C++ development, but using `#defines` for Unicode and other string constants have the advantage that they can be automatically concatenated, at compile time, without having to clutter your code with concatenation operators.

Listing 1: *Unicode in C*

```
// Various character and string types
wchar_t *wchar_msg =
  L"Platform-specific-sized text (UTF-16 on Windows)";
char16_t *utf16_msg =
  u"Cross-platform UTF-16 text, new with C++0x";
char32_t *utf32_msg =
  U"Cross-platform UTF-32 text, new with C++0x";
#if 0
char *utf8_msg =
  u8"UTF-8 text, unsupported by C++Builder";
#endif

// A smiley face as a Unicode escape code
wchar_t *msg1 = L"Hello, world! \u263A\n";

// A smiley face using a preprocessor macro and string
// concatenation.
#define SMILEY_FACE L"\u263A"
wchar_t *msg2 =
  L"Hello, world! " SMILEY_FACE "\n";
// In a real project, such #defines would probably go
// in their own project-wide header file.
```

Listing 1 shows examples of Unicode literals, escape codes, and preprocessor macros.

Working with Unicode in C

C developers are used to using <string.h> functions such as `strlen()`, `strcpy()`, and `strcat()` to manipulate C-style strings. There are corresponding functions for manipulating C-style `wchar_t` strings; most `wchar_t` functions are defined both in <wchar.h> and in the "traditional" header file (<string.h> for plain string manipulation, <stdio.h> for I/O, etc.).

- For `wchar_t` string manipulation, use `wcslen()`, `wcscpy()`, `wcscat()`, and so on. (Replace "str" with "wcs.")
- For `wchar_t` file I/O, use functions like `fgetws()` and `fputwc()` instead of `fgets()` and `fputc()`. (Insert "w" before the data type.)
- `printf()`, `sscanf()`, and so on become `wprintf()`, `swscanf()`, and so on. (Insert "w" before "printf" or "scanf.") Take note to not confuse `swprintf()` (wide character `sprintf()`) with `wsprintf()` (the Windows implementation of `sprintf()`).
- File and directory manipulation functions, such as `fopen()`, `opendir()`, `mkdir()`, and `_unlink()`, become `_wfopen()`, `_wopendir()`, `_wmkdir()`, and `_wunlink()`. (Add "_w" to the beginning.) These let you manipulate files and directories with Unicode characters in their names.

The `printf()` and `scanf()` family deserve special mention. `wprintf()` and `wscanf()` act like their narrow character counterparts in most respects, but the format strings needed to specify char and char * arguments changes for wide characters (and also differs between compilers). For example, Visual C++ and C++Builder's implementations of `wprintf()` and `wscanf()` interpret `"%s"` as `wchar_t`, while GCC's runtime library on Linux interprets `"%s"` as `char` (just like `printf()` and `sscanf()`). (For vender references and the C99 draft standard's take on this, see [7], [8], and [9].)

Table 1 summarizes how format specifiers are interpreted for different platforms. The fact that C++Builder's treatment of format specifiers depends on the function being called, combined with this inconsistency in how specifiers are handled between compilers, can be a source of confusion, so be careful.

Platform	`"%c"` or `"%s"`	`"%lc"` or `"%ls"`	`"%hc"` or `"%hs"`
C99	`char`	`wchar_t`	undefined
Windows (VC++ and C++Builder)	`wchar_t`	`wchar_t`	`char`
Linux (GCC)	`char`	`wchar_t`	`char`

Table 1: *Format specifiers for wprintf and wscanf.*

These C string `wchar_t` functions are inconsistently documented; the wchar.h topic in the RAD Studio 2009 Documentation (which is more conveniently organized than RAD Studio 2010's documentation in this case) has a semi-complete listing, or you can browse the include files yourself.

Unicode in C++

String manipulation in C++ generally involves the use of the `std::string` class, as well as the various <iostream> classes for input, output, and string buffering. Developers familiar with Boost may also uses classes such as `boost::regex` or `boost::format` to help with string manipulation.

As it turns out, switching to the `wchar_t` version of these classes is quite easy: just prefix a w to each

Listing 2: *Unicode in C++*

```
#include <tchar.h>
#include <string>
#include <iostream>
#include <sstream>

using namespace std;

int _tmain(int, _TCHAR*)
{
  int value;

  // ANSI (narrow character) code
  string s1 = "123";
  stringstream stream1(s1);
  stream1 >> value;
  cout << "The value is " << value << endl;

  // Wide character version. Easy!
  wstring s2 = L"124";
  wstringstream stream2(s2);
  stream2 >> value;
  wcout << L"The value is " << value << endl;

  return 0;
}
```

class name. See **Listing 2** for sample char string code in C++ and its corresponding wchar_t code.

Most text-related classes (including <iostream>) in the C++ Standard Library and in Boost are actually typedefs for template classes. For example, std::string is actually a typedef for std::basic_string<char>, and std::wstring is a typedef for std::basic_string<wchar_t>. Because std::basic_string is a template, it can be instantiated on any char-like data type that you wish. This means that, if you need to work with C++0x's char16_t or char32_t data types, you can use std::basic_string<char16_t> instead of std::string, use std::basic_fstream<char32_t> instead of std::fstream, and so on.

Unicode in the VCL

This is where things get interesting. Starting with RAD Studio 2009, the VCL offers several string classes which support ANSI, UTF-8, and UTF-16 encodings:

- AnsiString corresponds to the old String class. It contains 8-bit (char) data in the system default code page.
- UnicodeString is the new class, containing 16-bit (wchar_t) data in the UTF-16 encoding.
- WideString still exists from previous versions of RAD Studio. It corresponds to COM's BSTR data type and contains 16-bit (wchar_t) data like UnicodeString. Because UnicodeString uses C++Builder's own memory management and reference counting, it's often faster than WideString, so unless you need easy interoperability with COM, you should use the new UnicodeString class.
- AnsiStringT is a class template that contains 8-bit (char) data encoded in *any* code page; the code page is given as the template parameter. (AnsiString is actually a typedef for AnsiStringT<0>.) The requirement that the code page be given as a template parameter prevents you from using AnsiStringT with arbitrary code pages at runtime, so if you need that capability, you may need to instead use RawByteString (below) or use one of the C or C++ string manipulation methods instead of using the VCL.
- UTF8String is an AnsiStringT instantiation using the UTF-8 encoding.
- RawByteString contains 8-bit (char) data in an *unspecified* code page. The VCL will avoid applying any code page conversions to RawByteStrings; it becomes the calling code's responsibility to correctly handle code pages issues. Using RawByteString can have several advantages: since each code page is otherwise a separate compile-time type, RawByteString lets you write a single routine that can handle any code page; it removes any VCL overhead of doing code page conversions itself; and it prevents possible loss of data from automatically converting text data into encodings that can't represent some characters.

Most member functions of these new string classes operate just the same as they did for the old pre-C++Builder 2009 String class. The printf-type methods (printf(), sprintf(), vprintf(), cat_printf(), cat_sprintf(), and cat_sprintf()) deserve special mention. Like C's wprintf() and wscanf() functions, their treatment of the "%s" and "%c" format specifiers depends on whether they're called on an AnsiString or UnicodeString instance. Refer back to **Table 1** for details.

Unicode in the Windows API

The Windows API includes both Unicode and ANSI variants. For example, the MessageBox function is actually two different functions: MessageBoxA, which takes ANSI strings, and MessageBoxW, which takes wide (UTF-16) strings. MessageBox itself is a macro that resolves to MessageBoxA or MessageBoxW depending on your preprocessor macros and project options. You're also free to explicitly call one API variant or the other, regardless of your project options, simply by calling MessageBoxA or MessageBoxW directly. Other Windows API functions dealing with text or string data have similar variants.

Which variant you get is determined by whether or not the UNICODE preprocessor macro is defined. In C++Builder, this macro is automatically defined depending on your project's options. To change this option from the IDE, go under the Project menu, under Options, under the top-level Directories and Conditions category, and examine the "_TCHAR maps to" option. If it's set to "char," then the UNICODE preprocessor macro is left undefined and the ANSI variant of the Windows API is used. If it's set to "wchar_t," then the UNICODE macro is defined and the wide-string

(UTF-16) variant of the Windows API is used.

The `UNICODE` macro also affects the use of tchar.h in writing code that can compile as ANSI or Unicode. This will be discussed in Part II.

The Windows API also includes functions such as `CharNext()`, `CharPrev()`, and `CompareString()` that are capable of dealing with complexities such as composite characters and surrogate pairs. See MSDN [10] for details.

Conclusion

In this article, I provided an introduction to Unicode and I presented an overview of how to use Unicode in C, C++, the VCL, and the Windows API. For a more detailed introduction to Unicode, see [11].

In Part II of this series, I'll show you how to migrate your existing C++Builder applications to use Unicode.

Contact Josh at **joshkel@gmail.com**.

References

1. Joel Spolsky, "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)." http://www.joelonsoftware.com/articles/Unicode.html
2. Wikipedia, "Unicode." http://en.wikipedia.org/wiki/Unicode
3. The Unicode Consortium. "Glossary." http://unicode.org/glossary/
4. "ICU – International Components for Unicode." http://site.icu-project.org/
5. "How to enter Unicode characters in Microsoft Windows." http://www.fileformat.info/tip/microsoft/enter_unicode.htm
6. "Unicode." http://www.fileformat.info/info/unicode/index.htm
7. "MSDN: Size and Distance Specification." http://msdn.microsoft.com/en-us/library/tcxf1dw6%28v=VS.80%29.aspx.
8. "Linux Programmer's Manual: printf(3)." http://www.kernel.org/doc/man-pages/online/pages/man3/printf.3.html. Retrieved 4/12/2010.
9. "WG14/N1124 Committee Draft." http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf. Retrieved 4/12/2010.
10. "MSDN: String Reference: Functions." http://msdn.microsoft.com/en-us/library/ff468910%28VS.85%29.aspx. Retrieved 4/14/2010.
11. "Internationalization for Windows Applications (Windows)." http://msdn.microsoft.com/en-us/library/dd318661%28VS.85%29.aspx.

Get over 12 years of the Journal on CD!

Version 4.0 of our popular **Archive CD** contains over 12 years of the C++Builder Developer's Journal (from June 1997 through December 2008), all neatly presented through an easy-to-use HTML user interface.

You can view the CD contents with any browser and PDF reader.

Includes all article **source code** too!

With **hundreds of articles**, **illustrations,** and **source code** examples, this is the most complete set of information about C++Builder that you can find in one place!

Order: To order online, visit http://bcbjournal.org/archive_cd.htm

Price: Each CD is **$39.00** plus shipping.

Package: For just **$79.00**, get both a CD and a one-year subscription.