XPCOM:Strings

From MDC

Contents

[hide]

Preface

[edit]

by Alec Flett

Thanks to David Baron for actual docs ,
Peter Annema for lots of direction,
Myk Melez for some more docs, and
David Bradley for a diagram
Revised by Darin Fisher for Mozilla 1.7
Revised by Jungshik Shin to clarify character encoding issues

This guide will attempt to document the plethora of string classes, and hopefully provide an answer to the age old question, "what string class should I use here?"

If you are a Mozilla embedder or if you are writing an XPCOM component that will be distributed separately from the Mozilla code base, then this string guide is most likely not for you! Provided you are developing against Mozilla 1.7 or later, you should instead be using the new minimal Mozilla string API and in particular the nsEmbedString class.

In a hurry? Go check out the String Quick-Reference ([1]).

Introduction [edit]

The string classes are a library of C++ classes which are used to manage buffers of unicode and single-byte character strings. They reside in the mozilla codebase in the `xpcom/string` directory.

Abstract (interface) classes begin with "nsA" and concrete classes simply begin with "ns". Classes with a "`CString`" in the name store 8-bit bytes (`char`'s) which may refer to single byte ASCII strings, or multibyte Unicode strings encoded in UTF-8 or a (multibyte or single byte) legacy character encoding (e.g. ISO-8859-1, Shift_JIS, GB2312, KOI8-R). All other classes simply have "`String`" in their name and refer to 16-bit strings made up of `PRUnichar`'s, For example: `nsAString` is an abstract class for storing Unicode characters in UTF-16 encoding, and `nsDependentCString` is a concrete class which stores a 8-bit string. Every 16-bit string class has an equivalent 8-bit string class. For example: `nsCString` is the 8-bit string class which corresponds to `nsString`.

8-bit and 16-bit string classes have completely separate base classes, but share the same APIs. As a result, you cannot assign a 8-bit string to a 16-bit string without some kind of conversion helper class or routine. For the purpose of this document, we will refer to the 16-bit string classes in class documentation. It is safe to assume that every 16-bit class has an equivalent 8-bit class.

String Guidelines [edit]

Follow these simple rules in your code to keep your fellow developers, reviewers, and users happy.

- Avoid `*WithConversion` functions at all costs: `AssignWithConversion`, `AppendWithConversion`, `EqualsWithConversion`, etc
- Use the most abstract string class that you can. Usually this is:

        - nsAString for function parameters
        - nsString for member variables
        - nsAutoString or nsXPIDLString for local (stack-based) variables
- Use NS_LITERAL_[C]STRING / NS_NAMED_LITERAL_[C]STRING to represent literal strings (i.e. "foo") as nsAString-compatible objects.
- Use string concatenation (i.e. the "+" operator) when combining strings.
- Use nsDependentString when you have a raw character pointer that you need to convert to an nsAString-compatible string.
- Use Substring() to extract fragments of existing strings.
- Use iterators to parse and extract string fragments.

The Abstract Classes [edit]

Every string class derives from `nsAString` (or `nsACString`). This class provides the fundamental interface for access and manipulation of strings. While concrete classes derive from `nsAString`, `nsAString` itself cannot be instantiated.

This is very similar to the idea of an "interface" that mozilla uses to describe abstract object descriptions in the rest of the codebase. In the case of interfaces, class names begin with "nsI" where "I" refers to "Interface". In the case of strings, abstract classes begin with "nsA" and the "A" means "Abstract".

There are a number of abstract classes which derive from `nsAString`. These abstract subclasses also cannot be instantiated, but they describe a string in slightly more detail than `nsAString`. They guarantee that the underlying implementation behind the abstract class provides specific capabilities above and beyond `nsAString`.

The list below describes the main base classes. Once you are familiar with them, see the appendix describing What Class to Use When.

- **`nsAString`**: the abstract base class for all strings. It provides an API for assignment, individual character access, basic manipulation of characters in the string, and string comparison. This class corresponds to the XPIDL `AString` parameter type.
- **`nsSubstring`**: the common base class for all of the string classes. Provides optimized access to data within the string. A `nsSubstring` is not necessarily null-terminated. (For backwards compatibility, `nsASingleFragmentString` is a typedef for this string class.)
- **`nsString`**: builds on `nsSubstring` by guaranteeing a null-terminated storage. This allows for a method (`.get()`) to access the underlying character buffer. (For backwards compatibility, `nsAFlatString` is a typedef for this string class.)

The remainder of the string classes inherit from either `nsSubstring` or `nsString`. Thus, every string class is compatible with `nsAString`.

It's important to note that `nsSubstring` and `nsAString` both represent a contiguous array of characters that are not necessarily null-terminated. One might ask then ask why two different yet similar string classes need to exist. Well, `nsSubstring` exists primarily as an optimization since `nsAString` must retain binary compatibility with the frozen `nsAString` class that shipped with Mozilla 1.0. Up until the release of Mozilla 1.7, `nsAString` was capable of representing a string broken into multiple fragments. The cost associated with supporting multi-fragment strings was high and offered limited benefits. It was decided to eliminate support for multi-fragment strings in an effort to reduce the complexity of the string classes and improve performance. See bug 231995 for more details.

Though `nsSubstring` provides a more efficient interface to its underlying buffer than `nsAString`, `nsAString` is still the most commonly used class for parameter passing. This is because it is the string class corresponding to `AString` in XPIDL. Therefore, this string guide will continue to discuss the string classes with an emphasis on `nsAString`.

Since every string derives from `nsAString` (or `nsACString`), they all share a simple API. Common read-only methods:

- **`.Length()`** - the number of code units (bytes for 8-bit string classes and PRUnichar's for 16-bit string classes) in the string.
- **`.IsEmpty()`** - the fastest way of determining if the string has any value. Use this instead of testing `string.Length` == 0
- **`.Equals(string)`** - TRUE if the given string has the same value as the current string.

Common methods that modify the string:

- **`.Assign(string)`** - Assigns a new value to the string.
- **`.Append(string)`** - Appends a value to the string.
- **`.Insert(string, position)`** - Inserts the given string before the code unit at position.
- **`.Truncate(length)`** - shortens the string to the given length.

Complete documentation can be found in the Appendix.

Read-only strings

[edit]

The `const` attribute on a string determines if the string is writable. If a string is defined as a `const nsAString` then the data in the string cannot be manipulated. If one tries to call a non-`const` method on a `const` string the compiler will flag this as an error at build time.

For example:

```
void nsFoo::ReverseCharacters(nsAString& str) {
      ...
     str.Assign(reversedStr); // modifies the string
}
```

This should not compile, because you're assigning to a `const` class:

```
void nsFoo::ReverseCharacters(const nsAString& str) {
      ...
     str.Assign(reversedStr);
}
```

As function parameters

[edit]

It is recommended that you use the most abstract interface possible as a function parameter, instead of using concrete classes. The convention is to use C++ references (the '&' character) instead of pointers (the '*' character) when passing string references around. For example:

```
// abstract reference
nsFoo::PrintString(const nsAString& str) {..}

// using a concrete class!
nsFoo::PrintString(const nsString& str) {..}

// using a pointer!
nsFoo::PrintString(const nsAString* str) {..}
```

The abstract classes are also sometimes used to store temporary references to objects. You can see both of these uses in Common Patterns, below.

The Concrete Classes - which classes to use when [edit]

The concrete classes are for use in code that actually needs to store string data. The most common uses of the concrete classes are as local variables, and members in classes or structs. Whereas the abstract classes differ in storage mechansim, for the most part the concrete classes differ in storage policy.

The following is a list of the most common concrete classes. Once you are familiar with them, see the appendix describing What Class to Use When.

- **`nsString / nsCString`**- a null-terminated string whose buffer is allocated on the heap. Destroys its buffer when the string object goes away.
- **`nsAutoString / nsCAutoString`**- derived from `nsString`, a string which owns a 64 code unit buffer in the same storage space as the string itself. If a string less than 64 code units is assigned to an `nsAutoString`, then no extra storage will be allocated. For larger strings, a new buffer is allocated on the heap.
- **`nsXPIDLString / nsXPIDLCString`**- derived from `nsString`, this class supports the `getter_Copies()` operator which allows easy access to XPIDL `out wstring / string` parameters. This class also supports the notion of a null-valued buffer, whereas `nsString`'s buffer is never null.
- **`nsDependentString`**- derived from `nsString`, this string does *not* own its buffer. It is useful for converting a raw string (`const PRUnichar*` or `const char*`) into a class of type `nsAString`.
- **`nsPrintfCString`**- derived from `nsCString`, this string behaves like an `nsCAutoString`. The constructor takes parameters which allows it to construct a 8-bit string from a `printf`-style format string and parameter list.
- **`NS_LITERAL_STRING/NS_NAMED_LITERAL_STRING`**- these convert a literal string (such as "abc") to a `nsString` or a subclass of `nsString`. On platforms supporting double-byte string literals (e.g., MSVC++ or GCC with the -fshort-wchar option), these are simply macros around the `nsDependentString` class. They are slightly faster than just wrapping them with an `nsDependentString` because they use the compiler to calculate their length, and they also hide the messy cross-platform details of non-byte literal strings.

There are also a number of concrete classes that are created as a side-effect of helper routines, etc. You should avoid direct use of these classes. Let the string library create the class for you.

- **`nsSubstringTuple`** - created via string concatenation
- **`nsDependentSubstring`** - created through Substring
- **`nsPromiseFlatString`** - created through **`PromiseFlatString()`**

Of course, there are times when it is necessary to reference these string classes in your code, but as a general rule they should be avoided.

Iterators

[edit]

Iterators are objects that retain a reference to a position in a string. In some ways they are like a number which refers to an index in an array, or a character-pointer that refers to a position in a character string. They also provide a syntactic means to distinguish between reading and writing to a string.

Iterators are most often used to extract substrings of a string. They provide the capability to modify the contents of a string, but often helper routines, or the string's own methods are quicker at complex string transformations.

Iterators are declared from the string class which they are iterating:

```
nsAString::const_iterator start, end; // reading-only iterators for nsAString
nsString::iterator substr_start, substr_end; // writing iterators for nsString
```

Iterators are initialized with one of 4 methods on the string you wish to reference:

```
// let's read from 'str'
str.BeginReading(start); // initialize 'start' to the beginning of 'str'
str.EndReading(end); // 'end' will be at the end of the string

// say we also want to write to 'url'
url.BeginWriting(substr_start);
url.EndWriting(substr_end);
```

You can access the code unit that an iterator points to with the dereference operator *.

```
if (*start == '[')
     printf("Starts with a bracket\n");
```

Note in the above examples, that '`end`' and '`substr_end`' will actually point to the code unit past the end of the string, so you should never dereference the direct result of `.EndReading()`.

You can test if two iterators point to the same position with == or !=. You can advance iterators with ++. Putting the ++ before your iterator is preferred, and will prevent creation of a temporary iterator.

```
while (start != end) // iterate through the whole string
     ++start;
```

You can effectively write to a string with writing iterators (as opposed to const-iterators):

```
// change all * to !
while (substr_start != substr_end) {
     if (*substr_start == '*')
           *substr_start = '!';
     ++substr_start;
}
```

With the patch for bug 231995 , this loop is now as efficient as iterating with raw character pointers.

Helper Classes and Functions [edit]

Searching strings - looking for substrings, characters, etc. [edit]

`FindInReadable()` is the replacement for the old `string.Find(..)`. The syntax is:

```
PRBool FindInReadable(const nsAString& pattern,
                      nsAString::const_iterator start, nsAString::const_iterator end,
                      nsStringComparator& aComparator = nsDefaultStringComparator());
```

To use this, `start` and `end` should point to the beginning and end of a string that you would like to search. If the search string is found, `start` and `end` will be adjusted to point to the beginning and end of the found pattern. The return value is PR_TRUE or PR_FALSE, indicating whether or not the string was found.

An example:

```
const nsAString& str = GetSomeString();
nsAString::const_iterator start, end;

str.BeginReading(start);
str.EndReading(end);

NS_NAMED_LITERAL_STRING(valuePrefix, "value=");

if (FindInReadable(valuePrefix, start, end)) {
    // end now points to the character after the pattern
    valueStart = end;

}
```

Memory Allocation - how to avoid it, which methods to use [edit]

The preferred method to allocate a new character buffer (`PRUnichar*`/`char*`) from an existing string is with one of the following methods:

- **`PRUnichar* ToNewUnicode(`*`nsAString&`*`)`** - Allocates a `PRUnichar*`buffer from an `nsAString`.
- **`char *ToNewCString(`*`nsACString&`*`)`** - Allocates a `char*`buffer from an `nsACString`. Note that this method will also work on nsAStrings, but it will do an implicit lossy conversion. This function should only be used if the input is known to be strictly ASCII. Often a conversion to UTF-8 is more appropriate. See **`ToNewUTF8String`** below.
- **`char* ToNewUTF8String(`*`nsAString&`*`)`** - Allocates a new `char*` buffer containing the UTF-8 encoded version of the given nsAString. See Unicode Conversion for more details.

These methods return a buffer allocated using XPCOM's allocator (`nsMemory::Alloc`) instead of the traditional allocator (`malloc`, etc.). You should use `nsMemory::Free` to deallocate the result when you no longer need it.

Substrings (string fragments) [edit]

It is very simple to refer to a substring of an existing string without actually allocating new space and copying the characters into that substring. `Substring()` is the preferred method to create a reference to such a string.

```
void ProcessString(const nsAString& str) {
    const nsAString& firstFive = Substring(str, 0, 5);
    // firstFive is now a string representing the first 5 characters
}
```

Unicode Conversion ns*CString vs. ns*String [edit]

Strings can be *stored* in two basic formats: 8-bit code unit (byte/`char`) strings, or 16-bit code unit (`PRUnichar`) strings. Any string class with a capital "C" in the classname contains 8-bit bytes. These classes include `nsCString`, `nsDependentCString`, and so forth. Any string class *without* the "C" contains 16-bit code units.

A 8-bit string can be in one of many character encodings while a 16-bit string is always in UTF-16. The most common encodings are:

- ASCII - 8-bit encoding for basic English-only strings. Each ASCII value is stored in exactly one byte in the array.
- UCS2 - 16-bit encoding for a *subset* of Unicode, BMP . The Unicode value of a character stored in UCS2 is stored in exactly one 16-bit `PRUnichar` in a string class.
- UTF-8 - 8-bit encoding for Unicode characters. Each Unicode characters is stored in up to 4 bytes in a string class. UTF-8 is capable of representing the entire Unicode character repertoire, and it efficiently maps to UTF-32 .
- UTF-16 - 16-bit encoding for Unicode storage, backwards compatible with UCS2. The Unicode value of a character stored in UTF-16 may require *one or two* 16-bit `PRUnichars` in a string class. The contents of `nsAString` always has to be regarded as in this encoding instead of UCS2. UTF-16 is capable of representing the entire Unicode character repertoire, and it efficiently maps to UTF-32. (Win32 W APIs and Mac OS X natively use UTF-16.)

In addition, there are literally hundreds of encodings that are provided by internationalization libraries. Access to these libraries may be part of the application (such as `nsICharsetConversionManager` in Mozilla) or built into the operating system (such as `iconv()` in UNIX operating systems and `MultiByteToWideChar`/`WideCharToMultiByte` on Windows).

When working with existing code, it is important to examine the current usage of the strings that you are manipulating, to determine the correct conversion mechanism.

When writing new code, it can be confusing to know which storage class and encoding is the most appropriate. There is no single answer to this question, but there are a few important guidelines:

- **Is the string always ASCII?** First and foremost, you need to determine what kinds of values will be stored in the string. If the strings are always internal, ASCII strings such as "left", "true", "background" and so forth, then straight C-strings are probably the way to go.
- **If the string is ASCII, will it be compared to, assigned to, or otherwise interact with non-ASCII strings?** When assigning or comparing an 8-bit ASCII value (in)to a 16-bit UCS2 string, an "inflation" needs to happen at runtime. If your strings are small enough (say, less than 64 bytes) then it may make sense to store your string in a 16-bit unicode class as well, to avoid the extra conversion. The tradeoff is that your ASCII string takes up twice as much space as a 16-bit Unicode string than it would as an 8-bit string.
- **Is the string usually ASCII, but needs to support unicode?** If your string is most often ASCII but needs to be able to store Unicode characters, then UTF-8 may be the right encoding. ASCII characters will still be stored in 8-bit storage but other Unicode characters will take up 2 to 4 bytes. However if the string ever needs to be compared or assigned to a 16-bit string, a runtime conversion will be necessary.
- **Are you storing large strings of non-ASCII data?** Up until this point, UTF-8 might seem like the ideal encoding. The drawback is that for most non-European characters (such as Chinese, Indian and Japanese) in BMP, UTF-8 takes 50% more space than UTF-16. For characters in plane 1 and above, both UTF-8 and UTF-16 take 4 bytes.
- **Do you need to manipulate the contents of a Unicode string?** One problem with encoding Unicode characters in UTF-8 or other 8-bit storage formats is that the actual Unicode character can span multiple bytes in a string. In most encodings, the actual number of bytes varies from character to character. When you need to iterate over each character, you must take the encoding into account. This is vastly simplified when iterating 16-bit strings because each 16-bit code unit (`PRUnichar`) corresponds to a Unicode character as long as all characters are in BMP, which is often the case. However, you have to keep in mind that a single Unicode character in plane 1 and beyond is represented in two 16-bit code units in 16-bit strings so that the number of `PRUnichar`'s is *not* always equal to the number of Unicode characters. For the same reason, the position and the index in terms of 16-bit code units are not always the same as the position and the index in terms of Unicode characters.

To assist with ASCII, UTF-8, and UTF-16 conversions, there are some helper methods and classes. Some of these classes look like functions, because they are most often used as temporary objects on the stack.

UTF-8 / UTF-16 conversion

[edit]

`NS_ConvertUTF8toUTF16(`*`const nsACString&`*`)` - a `nsAutoString` subclass that converts a UTF-8 encoded `nsACString` or `const char*` to a 16-bit UTF-16 string. If you need a `const PRUnichar*` buffer, you can use the `.get()` method. For example:

```
/* signature: void HandleUnicodeString(const nsAString& str); */
object->HandleUnicodeString(NS_ConvertUTF8toUTF16(utf8String));

/* signature: void HandleUnicodeBuffer(const PRUnichar* str); */
object->HandleUnicodeBuffer(NS_ConvertUTF8toUTF16(utf8String).get());
```

`NS_ConvertUTF16toUTF8(`*`const nsAString&`*`)` - a `nsCAutoString` which converts a 16-bit UTF-16 string (`nsAString`) to a UTF-8 encoded string. As above, you can use `.get()` to access a `const char*` buffer.

```
/* signature: void HandleUTF8String(const nsACString& str); */
object->HandleUTF8String(NS_ConvertUTF16toUTF8(utf16String));

/* signature: void HandleUTF8Buffer(const char* str); */
object->HandleUTF8Buffer(NS_ConvertUTF16toUTF8(utf16String).get());
```

`CopyUTF8toUTF16(`*`const nsACString&, nsAString&`*`)` - converts and copies:

```
// return a UTF-16 value
void Foo::GetUnicodeValue(nsAString& result) {
    CopyUTF8toUTF16(mLocalUTF8Value, result);
 }
```

`AppendUTF8toUTF16(`*`const nsACString&, nsAString&`*`)` - converts and appends:

```
// return a UTF-16 value
void Foo::GetUnicodeValue(nsAString& result) {
    result.AssignLiteral("prefix:");
    AppendUTF8toUTF16(mLocalUTF8Value, result);
}
```

`UTF8ToNewUnicode(`*`const nsACString&, PRUint32* aUTF16Count = nsnull`*`)` - allocates and converts (the optional parameter will contain the number of 16-byte units upon return, if non-null):

```
void Foo::GetUTF16Value(PRUnichar** result) {
    *result = UTF8ToNewUnicode(mLocalUTF8Value);
}
```

`CopyUTF16toUTF8(`*`const nsAString&, nsACString&`*`)` - converts and copies:

```
// return a UTF-8 value
void Foo::GetUTF8Value(nsACString& result) {
    CopyUTF16toUTF8(mLocalUTF16Value, result);
}
```

`AppendUTF16toUTF8(`*`const nsAString&, nsACString&`*`)` - converts and appends:

```
// return a UTF-8 value
void Foo::GetUnicodeValue(nsACString& result) {
    result.AssignLiteral("prefix:");
    AppendUTF16toUTF8(mLocalUTF16Value, result);
}
```

`ToNewUTF8String(`*`const nsAString&`*`)` - allocates and converts:

```
void Foo::GetUTF8Value(char** result) {
    *result = ToNewUTF8String(mLocalUTF16Value);
}
```

Lossy Conversion [edit]

The following should only be used when you can guarantee that the original string is ASCII. These helpers are very similar to the UTF-8 / UTF-16 conversion helpers above.

UTF-16 to ASCII converters [edit]

These converters are ***very dangerous*** because they ***lose information*** during the conversion process. You should ***avoid UTF-16 to ASCII conversions*** unless your strings are guaranteed to be ASCII. Each 16-bit code unit in 16-bit string is simply cast to an 8-bit byte, which means all Unicode character values above 0xFF are converted to an arbitrary 8-bit byte.

- **`NS_LossyConvertUTF16toASCII(`*`nsAString`*`)`** - a `nsCAutoString` which holds a temporary buffer containing the deflated value of the string.
- **`LossyCopyUTF16toASCII(`*`nsAString, nsACString`*`)`** - does an in-place conversion from UTF-16 into an ASCII string object.
- **`LossyAppendUTF16toASCII(`*`nsAString, nsACString`*`)`** - appends an UTF-16 string to an ASCII string, losing non-ASCII values.
- **`ToNewCString(`*`nsAString`*`)`** - allocates a new `char*` string.

ASCII to UTF-16 converters [edit]

These converters are ***very dangerous*** because they will ***mangle any non-ASCII string*** into a meaningless UTF-16 string. You should ***avoid ASCII to UTF-16 conversions*** unless your strings are guaranteed to be ASCII. For instance, if you have an 8-bit string encoded in a multibyte character encoding, each byte of the string will be "inflated" to a 16-bit number by simple casting.

For example, imagine a UTF-8 string where the first Unicode character of the string is represented with a 3-byte UTF-8 sequence, the "inflated" UTF-16 string will contain the 3 `PRUnichar`'s instead of the single `PRUnichar` that represents the first character. These `PRUnichar`'s have nothing to do with the first Unicode character in the UTF-8 string.

- **`NS_ConvertASCIItoUTF16(`*`nsACString`*`)`** - a `nsAutoString` which holds a temporary buffer containing the inflated value of the string.
- **`CopyASCIItoUTF16(`*`nsACString, nsAString`*`)`** - does an in-place conversion from one string into a Unicode string object.
- **`AppendASCIItoUTF16(`*`nsACString, nsAString`*`)`** - appends an ASCII string to a Unicode string.
- **`ToNewUnicode(`*`nsACString`*`)`** - Creates a new `PRUnichar*` string which contains the inflated value.

Common Patterns [edit]

Callee-allocated Parameters [edit]

Many APIs result in a method allocating a buffer in order to return strings to its caller. This can be tricky because the caller has to remember to free the string when they have finished using it. Fortunately, the `nsXPIDLString` class makes this very easy.

A method may look like this:

```
void GetValue(PRUnichar** aValue)
{
    *aValue = ToNewUnicode(foo);
}
```

Without the string classes, the caller would need to free the string:

```
{
    PRUnichar* val;
    GetValue(&val);

    if (someCondition) {
        // don't forget to free the value!
        nsMemory::Free(val);
        return NS_ERROR_FAILURE;
    }

    ...
    // and later, still don't forget to free!
    nsMemory::Free(val);
}
```

With `nsXPIDLString` you never have to worry about this. You can just use `getter_Copies()` to wrap the string class, and the class will remember to free the buffer when it goes out of scope:

```
{
    nsXPIDLString val;
    GetValue(getter_Copies(val));

    // val will free itself here
    if (someCondition)
        return NS_ERROR_FAILURE;
    ...
    // and later, still nothing to free
}
```

The resulting code is much simpler, and easy to read.

Literal Strings [edit]

A *literal string* is a raw string value that is written in some C++ code. For example, in the statement `printf("Hello World\n");` the value `"Hello World\n"` is a literal string. It is often necessary to insert literal string values when an `nsAString` or `nsACString` is required. These four macros will provide you with the necessary conversion:

- **`NS_LITERAL_CSTRING(`*`literal string`*`)`** - a temporary `nsCString`
- **`NS_NAMED_LITERAL_CSTRING(`*`variable,literal string`*`)`** - declares a `nsCString` variable named *variable*
- **`NS_LITERAL_STRING(`*`literal string`*`)`** - a temporary `nsString` with the unicode version of *literal string*
- **`NS_NAMED_LITERAL_STRING(`*`variable,literal string`*`)`** - declares a `nsString` variable named *variable* with the unicode version of *literal string*

The purpose of the `CSTRING` versions of these macros may seem unnecessary, given that `nsDependentCString` will also wrap a string value in an `nsCString`. The advantage to these macros is that the length of these strings is calculated at compile time, so the string does not need to be scanned at runtime to determine its length.

The `STRING` versions of these macros provide a portable way of declaring UTF-16 versions of the given literal string, avoiding runtime conversion on platforms which support literal UTF-16 strings (e.g., MSVC++ and GCC with the -fshort-wchar option).

```
// call Init(const PRUnichar*)
Init(L"start value"); // bad - L"..." is not portable!
Init(NS_ConvertASCIItoUTF16("start value").get()); // bad - runtime ASCII->UTF-16 conversion!

// call Init(const nsAString&)
Init(nsDependentString(L"start value")); // bad - not portable!
Init(NS_ConvertASCIItoUTF16("start value")); // bad - runtime ASCII->UTF-16 conversion!

// call Init(const nsACString&)
Init(nsDependentCString("start value")); // bad - length determined at runtime
```

Here are some examples of proper `NS_LITERAL_[C]STRING` usage.

```
// call Init(const PRUnichar*)
Init(NS_LITERAL_STRING("start value").get());

// call Init(const nsAString&)
Init(NS_LITERAL_STRING("start value"));

// call Init(const nsACString&)
Init(NS_LITERAL_CSTRING("start value"));
```

There are a few details which can be useful in tracking down issues with these macros:

`NS_LITERAL_STRING` does compile-time conversion to UTF-16 on some platforms (e.g. Windows, Linux, and Mac) but does runtime conversion on other platforms. By using `NS_LITERAL_STRING` your code is guaranteed to use the best possible conversion for the platform in question.

Because some platforms do runtime conversion, the use of literal string concatenation inside a `NS_LITERAL_STRING/NS_NAMED_LITERAL_STRING` macro will compile on these platforms, but not on platforms which support compile-time conversion.

For example:

```
// call Init(nsAString&)
Init(NS_LITERAL_STRING("start "
     "value")); // only compiles on some platforms
```

The reason for this is that on some platforms, the `L"..."` syntax is used, but it is only applied to the first string in the concatenation (`"start "`). When the compiler attempts to concatenate this with the non-Unicode string `"value"` it gets confused.

String Concatenation

[edit]

Strings can be concatenated together using the + operator. The resulting string is a `const nsSubstringTuple` object. The resulting object can be treated and referenced similarly to a `nsAString` object. Concatenation *does not copy the substrings*. The strings are only copied when the concatenation is assigned into another string object. The `nsSubstringTuple` object holds pointers to the original strings. Therefore, the `nsSubstringTuple` object is dependent on all of its substrings, meaning that their lifetime must be at least as long as the `nsSubstringTuple` object.

For example, you can use the value of two strings and pass their concatenation on to another function which takes an `const nsAString&`:

```
void HandleTwoStrings(const nsAString& one, const nsAString& two) {
    // call HandleString(const nsAString&)
    HandleString(one + two);
}
```

NOTE: The two strings are implicitly combined into a temporary `nsString` in this case, and the temporary string is passed into `HandleString`. If `HandleString` assigns its input into another `nsString`, then the string buffer will be shared in this case negating the cost of the intermediate temporary. You can concatenate N strings and store the result in a temporary variable:

```
NS_NAMED_LITERAL_STRING(start, "start ");
NS_NAMED_LITERAL_STRING(middle, "middle ");
NS_NAMED_LITERAL_STRING(end, "end");
// create a string with 3 dependent fragments - no copying involved!
nsString combinedString = start + middle + end;

// call void HandleString(const nsAString&);
HandleString(combinedString);
```

If you are using `NS_LITERAL_STRING` to create a temporary that is only used once, then it is safe to define it inside a concatenation because the string buffer will live as long as the temporary concatenation object (of type `nsSubstringTuple`).

```
// call HandlePage(const nsAString&);
// safe because the concatenated-string will live as long as its substrings
HandlePage(NS_LITERAL_STRING("start ") + NS_LITERAL_STRING("end"));
```

Local variables [edit]

Local variables within a function are usually stored on the stack. The `nsAutoString/nsCAutoString` classes are derivatives of the `nsString/nsCString classes`. They own a 64-character buffer allocated in the same storage space as the string itself. If the `nsAutoString` is allocated on the stack, then it has at its disposal a 64-character stack buffer. This allows the implementation to avoid allocating extra memory when dealing with small strings.

```
...
nsAutoString value;
GetValue(value); // if the result is less than 64 code units,
                 // then this just saved us an allocation
...
```

Member variables [edit]

In general, you should use the concrete classes `nsString` and `nsCString` for member variables.

```
class Foo {
    ...
    // these store UTF-8 and UTF-16 values respectively
    nsCString mLocalName;
    nsString mTitle;
};
```

Note that the strings are declared directly in the class, not as pointers to strings. Don't do this:

```
class Foo {
public:
    Foo() {
        mLocalName = new nsCString();
        mTitle = new nsString();
    }
    ~Foo() { delete mLocalName; delete mTitle; }

private:
    // these store UTF-8 and UTF-16 values respectively
    nsCString* mLocalName;
    nsString*  mTitle;
};
```

The above code may appear to save the cost of the string objects, but `nsString/nsCString` are small objects - the overhead of the allocation outweighs the few bytes you'd save by keeping a pointer.

Another common incorrect pattern is to use `nsAutoString/nsCAutoString` for member variables. As described in Local Variables, these classes have a built in buffer that make them very large. This means that if you include them in a class, they bloat the class by 64 bytes (`nsCAutoString`) or 128 bytes (`nsAutoString`).

An example:

```
class Foo {
    ...

    // bloats 'Foo' by 128 bytes!
    nsAutoString mLocalName;
};
```

Raw Character Pointers

[edit]

`PromiseFlatString()` can be used to create a temporary buffer which holds a null-terminated buffer containing the same value as the source string. `PromiseFlatString()` will create a temporary buffer if necessary. This is most often used in order to pass an `nsAString` to an API which requires a null-terminated string.

In the following example, an `nsAString` is combined with a literal string, and the result is passed to an API which requires a simple character buffer.

```
// Modify the URL and pass to AddPage(const PRUnichar* url)
void AddModifiedPage(const nsAString& url) {
    NS_NAMED_LITERAL_STRING(httpPrefix, "http://");
    const nsAString& modifiedURL = httpPrefix + url;

    // creates a temporary buffer
    AddPage(PromiseFlatString(modifiedURL).get());
}
```

`PromiseFlatString()` is smart when handed a string that is already null-terminated. It avoids creating the temporary buffer in such cases.

```
// Modify the URL and pass to AddPage(const PRUnichar* url)
void AddModifiedPage(const nsAString& url, PRBool addPrefix) {
    if (addPrefix) {
        // MUST create a temporary buffer - string is multi-fragmented
        NS_NAMED_LITERAL_STRING(httpPrefix, "http://");
        AddPage(PromiseFlatString(httpPrefix + modifiedURL));
    } else {
        // MIGHT create a temporary buffer, does a runtime check
        AddPage(PromiseFlatString(url).get());
    }
}
```

`printf` and a UTF-16 string [edit]

For debugging, it's useful to `printf` a UTF-16 string (nsString, nsAutoString, nsXPIDLString, etc). To do this usually requires converting it to an 8-bit string, because that's what printf expects. However, on Windows, the following should work:

```
printf("%S\n", yourString.get());
```

(Note: I didn't test this. Also, I'm not sure what exactly this does to non-ASCII characters, especially when they are outside the system codepage). The reason that this doesn't work on Unix is because a wchar_t, which is what %S expects, is usually 4 bytes there (even when Mozilla is compiled with -fshort-wchar, because this would require libc to be compiled with -fshort-wchar).

If non-ASCII characters aren't important, use:

```
printf("%s\n", NS_LossyConvertUTF16toASCII(yourString.get()));
```

On platforms that use UTF-8 for console output (most Linux distributions), this works:

```
printf("%s\n", NS_ConvertUTF16toUTF8(yourString.get()));
```

IDL [edit]

The string library is also available through IDL. By declaring attributes and methods using the specially defined IDL types, string classes are used as parameters to the corresponding methods.

IDL String types [edit]

The C++ signatures follow the abstract-type convention described above, such that all method parameters are based on the abstract classes. The following table describes the purpose of each string type in IDL.

IDL type	C++ Type	Purpose

`string`	`char*`	Raw character pointer to ASCII (7-bit) string, no string classes used. High bit is not guaranteed across XPConnect boundaries
`wstring`	`PRUnichar*`	Raw character pointer to UTF-16 string, no string classes used
`AString`	`nsAString`	UTF-16 string
`ACString`	`nsACString`	8-bit string, all bits are preserved across XPConnect boundaries
`AUTF8String`	`nsACString`	UTF-8 string - converted to UTF-16 as necessary when value is used across XPConnect boundaries
`DOMString`	`nsAString`	UTF-16 string used in the DOM. More or less the same as `AString`, but in JavaScript it has no distinction between whether the string is void or just empty. (not sure on this, looking for corrections.

C++ Signatures

[edit]

In IDL, `in` parameters are read-only, and the C++ signatures for `*String` parameters follows the above guidelines by using `const nsAString&` for these parameters. `out` and `inout` parameters are defined simply as `nsAString` so that the callee can write to them.

IDL	C++

`interface nsIFoo : nsISupports {`
 	
`    attribute AString utf16String;`
 	
`    AUTF8String getValue(in ACString key);`
`};`
	`class nsIFoo : public nsISupports {`
	
	`     NS_IMETHOD GetUtf16String(nsAString&`
	`                                aResult) = 0;`
	`     NS_IMETHOD SetUtf16String(const nsAString&`
	`                               aValue) = 0;`
	
	`     NS_IMETHOD GetValue(const nsACString& aKey,`
	`                         nsACString& aResult) = 0;`
	`};`

In the above example, `utf16String` is treated as a UTF-16 string. The implementation of `GetUtf16String()` will use `aResult.Assign` to "return" the value. In `SetUtf16String()` the value of the string can be used through a variety of methods including Iterators, `PromiseFlatString`, and assignment to other strings.

In `GetValue()`, the first parameter, `aKey`, is treated as a raw sequence of 8-bit values. Any non-ASCII characters in `aKey` will be preserved when crossing XPConnect boundaries. The implementation of `GetValue()` will assign a UTF-8 encoded 8-bit string into `aResult`. If the `this` method is called across XPConnect boundaries, such as from a script, then the result will be decoded from UTF-8 into UTF-16 and used as a Unicode value.

Choosing a string type

[edit]

It can be difficult to determine the correct string type to use for IDL. The following points should help determine the appropriate string type.

- Using string classes may avoid new memory allocation for `out` parameters. For example, if the caller is using an `nsAutoString` to receive the value for an `out` parameter, (defined in C++ as simply `nsAString&` then assignment of short (less than 64-characters) values to an `out` parameter will only copy the value into the `nsAutoString`'s buffer. Moreover, using the string classes allows for sharing of string buffers. In many cases, assigning from one string object to another avoids copying in favor of simply incrementing a reference count.
- `in` strings using string classes often have their length pre-calculated. This can be a performance win.
- In cases where a raw-character buffer is required, `string` and `wstring` provide faster access than `PromiseFlatString`.
- UTF-8 strings defined with `AUTF8String` may need to be decoded when crossing XPConnect boundaries. This can be a performance hit. On the other hand, UTF-8 strings take up less space for strings that are commonly ASCII.
- UTF-16 strings defined with `wstring` or `AString` are fast when the unicode value is required. However, if the value is more often ASCII, then half of the storage space of the underlying string may be wasted.

Appendix A - What class to use when

[edit]

This table provides a quick reference for what classes you should be using.

Context	class	Notes
Local Variables	`nsAutoString` `nsCAutoString`	
Class Member Variables	`nsString` `nsCString`	
Method Parameter types	`nsAString` `nsACString`	Use abstract classes for parameters. Use `const nsAString&` for "in" parameters and `nsAString&` for "out" parameters.
Retrieving "out" string/wstrings	`nsXPIDLString` `nsXPIDLCString`	Use `getter_Copies()`. Similar to `nsString / nsCString`.
Wrapping character buffers	`nsDependentString` `nsDependentCString`	Wrap `const char* / const PRUnichar*` buffers.
Literal strings	`NS_LITERAL_STRING` `NS_LITERAL_CSTRING`	Similar to `nsDependent[C]String`, but pre-calculates length at build time.

Appendix B - nsAString Reference

[edit]

Read-only methods.

- **`Length()`**
- **`IsEmpty()`**

- **`IsVoid()`** - XPConnect will convert void nsAStrings to JavaScript `null`.
- **`BeginReading(`*`iterator`*`)`**
- **`EndReading(`*`iterator`*`)`**
- **`Equals(`*`string[, comparator]`*`)`**
- **`First()`**
- **`Last()`**
- **`CountChar()`**
- **`Left(`*`outstring, length`*`)`**
- **`Mid(`*`outstring, position, length`*`)`**
- **`Right(`*`outstring, length`*`)`**
- **`FindChar(`*`character`*`)`**

Methods that modify the string.

- **`Assign(`*`string`*`)`**
- **`Append(`*`string`*`)`**
- **`Insert(`*`string`*`)`**
- **`Cut(`*`start, length`*`)`**
- **`Replace(`*`start, length, string`*`)`**
- **`Truncate(`*`length`*`)`**
- **`SetIsVoid(`*`state`*`)`** - XPConnect will convert void nsAStrings to JavaScript `null`.
- **`BeginWriting(`*`iterator`*`)`**
- **`EndWriting(`*`iterator`*`)`**
- **`SetCapacity()`**

Retrieved from "http://developer.mozilla.org/en/docs/XPCOM:Strings "