How to convert between different types of counted-string string types

devblogs.microsoft.com/oldnewthing/20240620-00

June 20, 2024

Raymond Chen

Last time, we learned about the sadness of treating counted strings as null-terminated strings. How can we escape this cycle of depression?

Almost all of the counted-string types have a way of creating an instance with a pointer and a count. The trick is to use it.

Class	Creator
`std::wstring`	`std::wstring(p, n)`
`HSTRING`	`WindowsCreateString(p, n, &s)`
`Platform::String`	`ref new String(p, n)`
`winrt::hstring`	`winrt::hstring(p, n)`
`BSTR`	`SysAllocStringLen(p, n)`
`_bstr_t`	`_bstr_t(SysAllocStringLen(p, n), FALSE)`
`CComBSTR`	`CComBSTR(n, p)`
`CStringW`	`CStringW(p, n)`

In most cases, the creator is fairly obvious. The one weirdo is `_bstr_t`, which has no built-in constructor for counted strings. Instead, you have to use the underlying `SysAllocStringLen` function to create the counted string, and then pass it to the constructor in attach mode (`bCopy = FALSE`).

Of course, once you get the counted string in, you need a way to get it back out.

Class	Pointer and length

`std::wstring`	`s.c_str(), s.size()`
`HSTRING`	`WindowsGetStringRawBuffer(s, &n), n`
`Platform::String`	`s->Data(), s->Length()`
`winrt::hstring`	`s.c_str(), s.size()`
`BSTR`	`s, SysStringLen(s)`
`_bstr_t`	`static_cast<wchar_t*>(s), s.length()`
`CComBSTR`	`static_cast<BSTR>(s), s.Length()`
`CStringW`	`static_cast<wchar_t const*>(s), s.GetLength()`

The `static_cast`s in the final three rows are typically not necessary if the compiler knows that it has to produce a `wchar_t const*`.

Unfortunately, combining these two tables gives you a horrible *n* × *n* matrix of possibilities. Fortunately, you can reduce this to linear complexity by using `std::wstring_view` as a common currency.

```
std::wstring_view to_wstring_view(std::wstring const& s)
{
    return s;
    // or: return { s.c_str(), s.size() };
}

std::wstring_view to_wstring_view(HSTRING s)
{
    UINT32 size;
    auto p = WindowsGetStringRawBuffer(s, &size);
    return { p, size };
}

std::wstring_view to_wstring_view(Platform::String^ const& s)
{
    return { s->Data(), s->Length() };
}

std::wstring_view to_wstring_view(winrt::hstring const& s)
{
    return { s.c_str(), s.size() };
}

std::wstring_view to_wstring_view(wchar_t*) = delete;

std::wstring_view bstr_to_wstring_view(BSTR s)
{
    return { s, SysStringLen(s) };
}

std::wstring_view to_wstring_view(_bstr_t const& s)
{
    return { s, s.length() };
}

std::wstring_view to_wstring_view(CComBSTR const& s)
{
    return { s, s.Length() };
}

std::wstring_view to_wstring_view(CStringW const& s)
{
    return { s, s.GetLength() };
}
```

For `std::wstring`, we can take advantage of the pre-existing `wstring_view` conversion operator.

Note that the `BSTR` version has to be named `bstr_to_wstring_view` because `BSTR` is a type alias for `wchar_t*`, so just calling it `to_wstring_view` would make it too easy to pass `wchar_t*`s that aren't really `BSTR`s. We delete the `to_wstring_view(wchar_t*)` overload so

that the compiler won't choose to convert the pointer to a `std::wstring` and then use the `std::wstring` overload, which would result in a dangling pointer.[1]

The other half is creating the counted string from a `wstring_view`.

```
template<typename T>
T size_check(std::wstring_view v)
{
    if (v.size() > (std::numeric_limits<T>::max)()) {
        throw std::bad_alloc();
    }
    return static_cast<T>(v.size());
}

template<typename T>
T from_wstring_view(std::wstring_view v) = delete;

template<>
std::wstring from_wstring_view<std::wstring>(std::wstring_view v)
{
    return std::wstring(v.data(), v.size());
}

template<>
HSTRING from_wstring_view(std::wstring_view v)
{
    HSTRING s;
    THROW_IF_FAILED(
        WindowsCreateString(v.data(),
                            size_check<UINT32>(v)));
    return s;
}

template<>
Platform::String^ from_wstring_view<Platform::String^>(std::wstring_view v)
{
    return ref new Platform::String(
        v.data(),
        size_check<UINT32>(v));
}

template<>
winrt::hstring from_wstring_view<winrt::hstring>(std::wstring_view v)
{
    return winrt::hstring(v.c_str(), size_check<uint32_t>(v));
}

template<>
BSTR from_wstring_view<BSTR>(std::wstring_view v)
{
    return ::SysAllocStringLen(v.data(), size_check<UINT>(v));
}

template<>
_bstr_t from_wstring_view<_bstr_t>(std::wstring_view v)
{
    return _bstr_t(
```

```
        ::SysAllocStringLen(v.data(), size_check<UINT>(v)),
        FALSE);
}

template<>
CComBSTR from_wstring_view<CComBSTR>(std::wstring_view v)
{
    return CComBSTR(size_check<int>(v), v.data());
}

template<>
CStringW from_wstring_view<CStringW>(std::wstring_view v)
{
    return CStringW(v.data(), size_check<int>(v));
}
```

You can now convert from one type to another through the pair of functions `to_wstring_view` and `from_wstring_view`.

```
// _bstr_t to winrt::hstring
auto s = from_wstring_view<winrt::hstring>(to_hstring_view(b));
```

You could wrap this in an adapter class, with a bunch of custom conversion operators:

```
struct counted_string_converter
{
    std::wstring_view v;

    template<typename T>
    counted_string_converter(T const& source) :
        v(to_wstring_view(source)) {}

    counted_string_converter(std::wstring_view source) :
        v(source) {}

    static counted_string_converter from_bstr(BSTR source)
    {
        return counted_string_converter(
            bstr_to_wstring_view(source));
    }

    template<typename T>
    operator T() const {
        return from_wstring_view<T>(v);
    }
};

winrt::hstring GetName();

void example()
{
    std::wstring s = counted_string_converter(GetName());
}
```

The danger with the `counted_string_converter` is that you might try to store it in an `auto`, which could produce dangling pointers.

```
winrt::hstring GetName();

void example()
{
    // Oops: Reference to destroyed temporary
    auto s = counted_string_converter(GetName());
}
```

All of this is really awkward, and the simplest approach might be to create custom case-by-case conversions. After all, it's unlikely that any one program is going to need all $n \times n$ conversions.

Bonus chatter: The `to_wstring_view` is handy if you want to insert these counted strings into a C++ stream.

```
BSTR s = ⟦something⟧

// This stops at the embedded null.
std::wcout << s;

// This prints the whole string.
std::wcout << bstr_to_wstring_view(s);
```

[1] If you really want to construct a `wstring_view` from a pointer to a null-terminated C-style string, then just use the `wstring_view` constructor.