Windows Implementation Library

Overview

Windows Implementation Library, or WIL, is a header-only library created to help make working with the Windows API more predictable and (hopefully) bug free.

A majority of functions are in either the `wil::` or `wistd::` namespace. `wistd::` is used for things that have an equivalent in STL's `std::` namespace but have some special functionality like being exception-free. Everything else is in `wil::` namespace.

The primary usages of WIL in our code so far are...

Smart Pointers

Inside wil/resource.h are smart pointer like classes for many Windows OS resources like file handles, socket handles, process handles, and so on. They're of the form `wil::unique_handle` and call the appropriate/matching OS function (like `CloseHandle()` in this case) when they go out of scope.

Another useful item is `wil::make_unique_nothrow()` which is analogous to `std::make_unique` (except without the exception which might help you integrate with existing exception-free code in the console.) This will return a `wistd::unique_ptr` (vs. a `std::unique_ptr`) which can be used in a similar manner.

Result Handling

To manage the various types of result codes that come back from Windows APIs, the file wil/result.h provides a wealth of macros that can help.

As an example, the method `DuplicateHandle()` returns a `BOOL` value that is `FALSE` under failure and would like you to `GetLastError()` from the operating system to find out what the actual result code is. In this circumstance, you could use the macro `RETURN_IF_WIN32_BOOL_FALSE` to wrap the call to `DuplicateHandle()` which would automatically handle this pattern for you and return the `HRESULT` equivalent on failure.

This leads to nice patterns where you can set up all resources in a function as protected by `std::unique_ptr` or the various `wil::` smart pointers and smart handles then `RETURN_IF_*` on every call to a Windows API and be guaranteed that your resources will be cleaned up appropriately under any failure case. Do note that this generally requires you to return an `HRESULT` as your return code and use out pointer parameters for return data. There are exceptions to this... read the header for more details.

The additional advantage to using this pattern is that failures at any point are logged to our global tracing/debugging channels to be viewed under the debugger

output with the exact line number and function details for the error.

Additionally, if you just want to make sure that a failure case is logged for debugging purposes, all of these macros have a `LOG_IF_*` equivalent that will simply log a failure and keep rolling.