Code Organization

Rules

- **Follow the pattern of what you already see in the code**
- Try to package new ideas/components into libraries that have nicely defined interfaces
- Package new ideas into classes or refactor existing ideas into a class as you extend
- Each project should have a Unit test in a ut_ folder in its subdirectory (like `ut_host`)
- Functional tests should be in ft_ subdirectories (like `ft_api`)
- Build scripts are generally in subdirectories with their type of output (like `/dll` or `/exe`)
- Try to place interfaces in an `inc` folder in an appropriate location
- Structure related libraries together (`/terminal/parser` and `/terminal/adapter`)

Code Overview

- `/` - root is where solution files, root MD documentation, SD replication artifacts go.
- `/bin` – not checked in is where binaries will be generated by the MSBuild system
- `/dep` – dependencies that aren't a part of the SDK
  - `/dep/console` – files that are currently in the internal-only private SDK for the console but we're working on opening
  - `/dep/DDK` – files lifted wholesale from the public Microsoft DDK so you don't have to install that. We're reducing dependencies on these but we still use TAEF (included in here) as our test runner engine
  - `/dep/NT` – some more structures from the DDK, sometimes internal-only/non-public we're trying to remove.
  - `/dep/telemetry` – Private Microsoft telemetry headers
  - `/dep/wil` – Windows Internal Library – extremely useful for interfacing with Win32/NT/COM APIs. Contains tons of "unique pointer" like syntax for various Win32 APIs and a handful of useful macros to enable cleaner code writing (RETURN_HR_IF, LOG_IF_WIN32_ERROR, etc.)
  - `/dep/win32k` – private headers from the Windows windowing system that we're trying to migrate off of
- `/ipch` – not checked in is where intellisense data will be generated if you use Visual Studio 2015
- `/obj` – not checked in is where objects will be generated by the MSBuild system
- `/src` – This is the fun one. In the root is common build system data.
  - `/src/cascadia` - This directory contains all the code specific to the Windows Terminal
    - `/src/cascadia/TerminalConnection` - This DLL is responsible for the various different ways a terminal instance can communicate with different terminal backends. Examples include the `ConptyConnection` (for communicating with Windows Console processes), or the `AzureCloudShellConnection` for communicating with Azure.
    - `/src/cascadia/TerminalSettings` - This is the DLL responsible for abstracting the settings for both the TerminalCore and the TerminalControl. This provides consumers of the TerminalControl a common interface for supplying settings to the Terminal.
    - `/src/cascadia/TerminalCore` - This LIB is responsible for the core implementation of a terminal instance. This defines one important class `Terminal` which is a complete terminal instance, with buffer, colors table, VT parsing, input handling, etc. It does *not* prescribe any sort of UI implementation - it should be connected to code that can handle rendering its contents, and provide input to it.

        - `/src/cascadia/TerminalControl` - This DLL provides the UWP-XAML implementation of a `TermControl`, which can be embedded within an application to provide a terminal instance within the application. It contains a DX renderer for drawing text to the screen, and translates input to send to the core Terminal. It also receives settings to apply to both itself and the core Terminal.
        - `/src/cascadia/TerminalApp` - This DLL represents the implementation of the Windows Terminal application. This includes parsing settings, hosting tabs & panes with Terminals in them, and displaying other UI elements. This DLL is almost entirely UWP-like code, and shouldn't be doing any Win32-like UI work.
        - `/src/cascadia/WindowsTerminal` - This EXE provides Win32 hosting for the TerminalApp. It will set up XAML islands, and is responsible for drawing the window, either as a standard window or with content in the titlebar (non-client area).
        - `/src/cascadia/CascadiaPackage` - This is a project for packaging the Windows Terminal and its dependencies into an .appx/.msix for deploying to the machine.
        - `/src/cascadia/PublicTerminalCore` - This is a DLL wrapper for the TerminalCore and Renderer, similar to `TermControl`, which exposes some exported functions that so the Terminal can be used from C#.
        - `/src/cascadia/WpfTerminalControl` - A DLL implementing a WPF version of the Terminal Control.
    - `/src/host` – The meat of the windows console host. This includes buffer, input, output, windowing, server management, clipboard, and most interactions with the console host window that aren't stated anywhere else. We're trying to pull things out that are reusable into other libraries, but it's a work in progress
        - `/src/host/lib` – Builds the reusable LIB copy of the host
        - `/src/host/dll` – Packages LIB into conhostv2.dll to be put into the OS C:\windows\system32\
        - `/src/host/exe` – Packages LIB into OpenConsole.exe currently used for testing without replacing your system32 copy
        - `/src/host/tools` – Random odds and ends that make testing/debugging/development a bit easier
            - ...to be doc'd, each of what these are
        - `/src/host/ut_host` – Contains complete unit test library for everything we've managed to get unit tests on. We'd like all new code to contribute appropriate unit tests in here
        - `/src/host/ft_api` – Feature level tests for anything that changes the way we interact with the outside world via the API. Building these up as we work as well
        - `/src/host/ft_cjk` – Double-wide/double-byte specific Chinese/Japanese/Korean language tests that previously had to be run in a different environment. To be merged into ft_api one day
        - `/src/host/ft_resize` – Special test for resizing/reflowing the buffer window
        - `/src/host/ft_uia` – Currently disabled (for not being very reliable) UI Automation tests that we are looking to re-enable and expand to do UI Automation coverage of various human interactions
        - `/src/host/...` - The files I'll list out below
    - `/src/inc` – Include files that are shared between the host and some of the other libraries. This is only some of them. The include story is kind of a mess right now, but we'd like to clean it up at some point

    - `/src/propslib` – Library shared between console host and the OS shell "right click a shortcut file and modify console properties" page to read/write user settings to and from the registry and embedded within shortcut LNK data
    - `/src/renderer` – Refactored extraction of all activities related to rendering the text in the buffers onto the screen
        - `/src/renderer/base` – Base interface layer providing non-engine-specific rendering things like choosing the data from the console buffer, deciding how to lay out or transform that data, then dispatching commands to a specific final display engine
        - `/src/renderer/gdi` – The GDI implementation of rendering to the screen. Takes commands to "draw a line" or "fill the background" or "select a region" from the base and turns them into GDI calls to the screen. Extracted from original console host code.
        - `/src/renderer/inc` – Interface definitions for all renderer communication
    - `/src/terminal` – Virtual terminal support for the console. This is the sequences that are found in-band with other text on STDIN/STDOUT that command the display to do things. This is the *nix way of controlling a console.
        - `/src/terminal/parser` – This contains a state machine and sorting engine for feeding in individual characters from STDOUT or STDIN and decoding them into the appropriate verbs that should be performed
        - `/src/terminal/adapter` – This converts the verbs from the interface into calls on the console API. It doesn't actually call through the API (for performance reasons since it lives inside the same binary), but it tries to remain as close to an API call as possible. There are some private extensions to the API for behaviors that didn't exist before this was written that we've not made public. We don't know if we will yet or force people to use VT to get at them.
    - `/src/tsf` – Text Services Foundation. This provides IME input services to the console. This was historically used for only Chinese, Japanese, and Korean IMEs specifically on OS installations with those as the primary language. It was in the summer of 2016 unrestricted to be able to be used on any OS installation with any IME (whether or not it will display correctly is a different story). It also was unrestricted to allow things like Pen and Touch input (which are routed via IME messages) to display properly inside the console from the TabTip window (the little popup that helps you insert pen/touch writing/keyboard candidates into an application)

Host File Overview

- Generally related to handling input/output data and sometimes intertwined with the actual service calls
    - `_output.cpp`
    - `_stream.cpp`
- Handles copy/paste/etc.
    - `clipboard.cpp`
- Handles the command prompt line as you see in CMD.exe (known as the processed input line… most other shells handle this themselves with raw input and don't use ours. This is a legacy of bad architectural design, putting stuff in conhost not in CMD)
    - `cmdline.cpp`
- Handles shunting IME data back and forth to the TSF library and to and from the various buffers
    - `Conimeinfo.cpp`
    - `Convarea.cpp`
- Contains the global state for the entire console application

    - `consoleInformation.cpp`
- Stuff related to the low-level server communication over our protocol with the driver
    - `Csrutil.cpp`
    - `Srvinit.cpp`
    - `Handle.cpp`
- Routines related to startup of the application
    - `Srvinit.cpp`
- Routines related to the API calls (and the servicing thereof, muxes a bit with the server protocol)
    - `Directio.cpp`
    - `Getset.cpp`
    - `Srvinit.cpp`
- Extra stuff strapped onto the buffer to enable CJK languages
    - `Dbcs.cpp`
- Attempted class-ification of existing Cursor structures to take them out of Screen Info/Text Info
    - `Cursor.cpp`
- Related to searching through the back buffer of the console (the output scroll buffer as defined in screeninfo/textinfo)
    - `Find.cpp`
- Contains global state data
    - `Globals.cpp`
- Attempted class-ification of existing Icon manipulation to take them out of ConsoleInformation/Settings
    - `Icon.cpp`
- Contains all keyboard/mouse input handling, capture of keys, conversion of keys, and some manipulation of the input buffer
    - `Input.cpp`
    - `Inputkeyinfo.cpp`
    - `Inputreadhandledata.cpp`
- Main entry point used ONLY by the OS to send a pre-configured driver handle to conhostv2.dll
    - `Main.cpp`
- Assorted utilities and stuff
    - `Misc.cpp` (left for us by previous eras of random console devs)
    - `Util.cpp` (created in our era)
- Custom zeroing and non-throwing allocator
    - `Newdelete.cpp`
- Related to inserting text into the TextInfo buffer
    - `Output.cpp`
    - `Stream.cpp`
- Connects to interfaces in the PropsLib to manipulate persistent settings state
    - `Registry.cpp`
- Connects to our relatively recently extracted renderer LIB to give it data about console state and user prefs
    - `renderData.cpp`
    - `renderFontDefaults.cpp`

- Maintains most of the information about what we should present inside the window on the screen (sizes, dimensions, also holds a text buffer instance and a cursor instance and a selection instance)
  - `screenInfo.cpp`
- Handles some aspects of scrolling with the mouse and keyboard
  - `Scrolling.cpp`
- Handles the click-and-drag highlighting of text on the screen to select (or the keyboard-based Mark mode selection where you can enter the mode and select around). Often calls clipboard when done
  - `Selection.cpp`
  - `selectionInput.cpp`
  - `selectionState.cpp`
- Handles all user preferences and state. Was extracted from consoleInformation and CI subclasses it still (because it was hard to break the association out)
  - `Settings.cpp`
- Good ol' Windows 10 telemetry pipeline & ETW events as debugging aids (they use the same channel with a different flag)
  - `Telemetry.cpp`
  - `Tracing.cpp`
- Private calls into the Windows Window Manager to perform privileged actions related to the console process (working to eliminate) or for High DPI stuff (also working to eliminate)
  - `Userprivapi.cpp`
  - `Windowdpiapi.cpp`
- New UTF8 state machine in progress to improve Bash (and other apps) support for UTF-8 in console
  - `Utf8ToWideCharParser.cpp`
- Window resizing/layout/management/window messaging loops and all that other stuff that has us interact with Windows to create a visual display surface and control the user interaction entry point
  - `Window.cpp`
  - `Windowproc.cpp`