P.A.L

product-oriented architecture layout

By Corey B.

This is document quickly describes a structured project layout rules that I use for all my projects (large and small) in C++

Why?

- I’ve come across my share of poorly organized large code bases. Inquiries behind the reasons for such a state of affairs often leads to one or more of the excuses below:
- (1) “We began this project when we were a start-up, so we had to create things fast and did not have time to design it right...”
- (2) “The guys that started this project did not have any experience with object oriented programming. They only knew how to code in C, Fortran and/or Matlab...”
- (3) “We inherited this code base from a different team / organization and have not had the time or resources to refactor it..”
- Over the years as I have maintained my own personal code base, I have developed a project layout that seems to work quite well for me. It all starts with simplification.

The Issues

- Some of the biggest issues with these code bases are:
- (1) Lack of documentation
- (2) They are usually tied to a ancient / proprietary development platforms and can not be updated without major refactoring
- (3) They use a third party dependencies that no longer have source available (publicly)
- (4) There is little to no organization of the source code. Everything is in the global name space
- (5) Inconsistent class/file names
- (6) An insane amount of pre and post build configuration steps
- (7) There is no concept of MVC in the code.
- (8) Liberal use of raw pointers in an age of smart pointers

The Solution?
A product-oriented architecture layout (PAL)

- Not a solution to all the issues listed but its a good start.
- What is it?
- A small list of simple rules to that can be applied to small to large scale C++ projects
- Less is more and simplicity trumps everything else
- Whether you have 1 product or 1000 products, the overall process architecture will remain the same. Project complexity **DOES NOT** have to increase as the amount of code increases.

Basic PAL structure

- All PAL project consists of three parts:
- Products (products) – A folder containing deployable projects and product-centered libraries.
- Commonly shared libraries (common) – A folder containing shared libraries and tools that are used exclusively by an project in the “Products” folder.
- Build configuration (CMakeLists.txt) – A simple script that defines how to build Products and Commonly shared items.

PAL Product Rules

- **†1.1** Products shall only depend on common libraries and/or other products.
- **†1.2** Each product folder shall manage its own resources, source code and documentation.
- **†1.3** Each product folder shall have its own “tests” sub-folder for managing unit-tests.
- **†1.4** Name-spaces shall match the folder layout for each class. Only the (inc, src) folders are ignored in name-space declarations.
- **†1.5** Sub folders may be used to group common product components and product libraries
- **†1.6** If applicable, the source and include files shall be in separate folders (inc, src)

PAL Common Rules

- **†2.1** Common libraries shall NOT depend on any code in the “products” folder.
- **†2.2** Common libraries shall have no dependencies(preferred) or only sibling dependencies within the common folder.
- **†2.3** Common libraries shall contain at least one dependency-free utility library.
- **†2.4** Common libraries shall contain at least one **contrib** folder that houses third-party libraries.
- **†2.5** All common libraries shall consist of one of the following (in order of preference):
  - **†2.5.1** Header only, Amalgamation code
  - **†2.5.2** Source code only library with build script (CMakeLists.txt)
  - **†2.5.3** Include files with pre-built binaries for each supported platform
- **†2.6** If applicable, the source and include files shall be in separate folders (inc, src)

PAL Code Rules

- **†3.1** There shall be a coding standard that is enforced by linters
- **†3.2** All public functions / member variables should have Doxygen supported comments in the header file.
- **†3.3** All header / source files should be named the same as their class names. (e.g: MyClass should be MyClass.h and MyClass.cpp)
- **†3.4** All code shall be self documenting with descriptive variable, function and class names
- **†3.5** Functions shall not exceed 300 lines of code
- **†3.6** If applicable, the source and include files shall be in separate folders (inc, src)

PAL Test Rules

- †4.1 Unit test projects shall exist for every project / library
- †4.2 Unit tests for common are preferred over for products
- †4.3 Unit tests shall test for expected good and bad functionality
- †4.4 Unit test projects should be enabled / disabled from the top level CMakeLists.txt

PAL Basic Project Rules

- **†5.1** Don’t use tools that encourage vendor lock-in or require that your code depend on some proprietary system or software. This could cause issues in the future.
- **†5.2** Create code that is cross-platform. Platform specific code should be used as a last resort.
- **†5.3** Choose simple, easily repetitive solutions rather than overly-complicated ones
- **†5.4** If possible avoid the use of large dependencies

Scaling Up

- As the project grows and more code / people are added to it, the layout remains simple and consistent.
- Conforming to these rules help keep the project from getting out of hand.
- I have been using this layout scheme for nearly 5 years and have over 100+ projects in my products folder with over 25+ common libraries. The complexity involved in adding, updating, building and deploying has remained constant.

The future

- I have actually started working on an IDE that is built around PAL. It will support everything from grabbing code from the SCM server to enabling localized continuous integration (more on this later) builds. An end to end solution that is simple and concise.

- The End