Canonical Project Structure

Document:	P1204R0
Audience:	SG15
Authors:	Boris Kolpackov (Code Synthesis)
Reply-To:	boris@codesynthesis.com
Date:	2018-10-08

Contents

1 Abstract
2 Background
3 Introduction
4 Source Directory
5 Source Naming
6 Source Contents
7 Tests
    7.1 Unit Tests
    7.2 Functional/Integration Tests
8 Acknowledgments

1 Abstract

We would like to propose source code layout and content guidelines for new C++ projects that would facilitate their packaging.

2 Background

The goal of establishing a canonical C++ project structure is to create an ecosystem of C++ packages that can coexist, are easy to comprehend by both humans and tools, scale to complex, real-world requirements, and, last but not least, are pleasant to work with.

The canonical structure is primarily meant for a *package* – a single library or program (or, sometimes, a collection of related programs) with a specific and well-defined function. While it may be less suitable for more elaborate, multi-library/program *end-products* that are not meant to be packaged, most of the recommendations discussed below would still apply. Specifically, we often find ourselves factoring common functionality out of such end-products and into separate packages, for example, in order to reuse it in other end-products. In this light, it may make sense to start a new end-product project as a composition of individual packages that follow the canonical structure.

Note also that these guidelines are intentionally minimal. They are only meant to cover the parts necessary to build a C++ project from source code and then validate the result (run tests). Specifically, we omit topics such as binary packages, examples, documentation, etc.

We also do not concern ourselves with actual packaging and dependency management. Rather, the goal is to make it easy for a project that follows the canonical structure to be built with any modern build system and packaged for any modern dependency manager. Specifically, we are not trying to describe a layout that could be handled fully automatically assuming that support for a new build system/dependency manager will still likely have to be added manually. However, the layout should be "sane enough" to make this task straightforward.

3 Introduction

The overall filesystem layout of a canonical project is presented below.

```
<name>/
├── <name>/
│   └── ...
└── tests/
    └── ...
```

This layout and its semantics are discussed in detail in the following sections with a short summary of the key points presented next.

- *Header and source files (or module interface and implementation files) are next to each other (no `include/` and `src/` split).*
- *Headers are included with `<>` and contain the project directory prefix, for example, `<libhello/hello.hpp>`.*
- *Header and source file extensions are `.hpp/.cpp` (`.mpp` for module interfaces).*
- *No special characters other than `_` and `-` in file names with `.` only used for extensions.*

As an example, the layout of a simple `libhello` library could look like this:

```
libhello/
├── libhello/
│   ├── hello.hpp
│   ├── hello.cpp
│   └── hello.test.cpp
└── tests/
    └── basics/
        └── driver.cpp
```

If a project consists of a library and an executable, then they should be split into separate projects. In this case, by convention, the library name should start with the `lib` prefix, for example, `libhello` and `hello`. Using the `lib` prefix for non-library projects should be avoided.

> An initial draft of these guidelines made the `lib` prefix for libraries a requirement. This, however, encountered strong push-back from early reviewers. Based on this feedback as well as to help with the adoption of these guidelines by existing projects, the `lib` prefix has been made optional.
>
> However, it is still strongly recommended to follow this convention in new projects since it offers several benefits:
>
> 1. It is clear from the name to both humans and tools what kind of project it is.
> 2. All libraries are consistently named (as opposed to some with the `lib` prefix and some without).
> 3. All library names are future-proofed to co-exist with executables. If one starts with a library without the `lib` prefix but later decides to add an executable, renaming the library would unlikely be an option. And there is no need to spend mental energy on thinking whether it's possible that an executable will be added later.

4 Source Directory

The project's source code is placed into a subdirectory of the root directory named the same as the project, for example, `hello/hello/` or `libhello/libhello/` . It is called the project's *source subdirectory*.

There are several reasons for this layout: It implements the canonical inclusion scheme (discussed below) where each header is prefixed with its project name. It also has a predictable name where users (and tools) can expect to find the project's source code. Finally, this layout prevents clutter in the project's root directory which usually contains various other files (like `README` , `LICENSE`) and directories (like `doc/` , `tests/` , `examples/`).

Another popular approach is to place public headers into the `include/` subdirectory and source files as well as private headers into `src/`. The cited advantage of this layout is the predictable location (`include/`) that contains only the project's public headers (that is, its API). This can make the project easier to navigate and understand while harder to misuse, for example, by including a private header.

However, this split layout is not without drawbacks:

- Navigating between corresponding headers and sources is cumbersome. This affects editing, grep'ing, as well as code browsing (for example, on GitHub).
- Implementing the canonical inclusion scheme would require an extra level of subdirectories (for example, `include/libhello/` and `src/libhello/`), which only amplifies the previous issue.
- Supporting generated source code can be challenging: Source code generators rarely provide support for writing headers and sources into different directories. Even if we can move things around post-generation, many build systems do not easily support this arrangement.

Also, the stated advantage of this layout – separation of public headers from private – is not as clear cut as it may seem at first. The common assumption of the split layout is that only headers from `include/` are installed and, conversely, to use the headers in-place, all one has to do is add `-I` pointing to `include/`. On the other hand, it is common for public headers to include private, for example, to call an implementation detail function in inline or template code (note that the same applies to private modules imported in public module interfaces). Which means such private, (or, probably now more accurately called *implementation detail*) headers have to be placed in the `include/` directory as well, perhaps into a subdirectory (such as `details/`) or with a file name suffix (such as `-impl`) to signal to the user that they are still "private". Needless to say, in an actively developed project, keeping track of which private headers can still stay in `src/` and which have to be moved to `include/` (and vice versa) is a tedious, error-prone task. As a result, practically, the split layout quickly degrades into the "all headers in `include/`" arrangement which negates its main advantage.

It is also not clear how the split layout will translate to modularized projects. With modules, both the interface and implementation (including non-inline/template

> function definitions) can reside in the same file with a substantial number of C++ developers finding this arrangement appealing. If a project consists of only such single-file modules, then `include/` and `src/` have effectively become the same thing (note that there couldn't be any "private" modules in `src/` since there would be nobody to import them). In a sense, we already have this situation with header-only libraries except that in the case of modules calling the directory `include/` would be an anachronism.
>
> To summarize, the split directory arrangement offers little benefit over the single directory layout, has a number of real drawbacks, and does not fit modularized projects well. In practice, private headers are placed into `include/`, often either in a subdirectory or with a special file name suffix, a mechanism that is readily available in the single directory layout.

All headers within a project should be included using the `<>` style inclusion and contain the project name as a directory prefix. And all headers means *all headers* – public, private, or implementation detail, in executables or in libraries.

As an example, let's say we've added `utility.hpp` to the `hello` executable project. This is how it should be included in `hello.cpp`:

```
// #include "utility.hpp"           // Wrong.
// #include <utility.hpp>           // Wrong.
// #include "../hello/utility.hpp"  // Wrong.

#include <hello/utility.hpp>
```

Similarly, if we want to include `hello.hpp` from `libhello`, then the inclusion should look like this:

```
#include <libhello/hello.hpp>
```

> The problem with the `""` style inclusion is if the header is not found relative to the including file, most implementations will continue looking for it in the include search paths, the same as for `<>`. As a result, if the header is not present in the right place (for example, because it was mistakenly not listed as to be installed), chances are that a completely unrelated header with the same name will be found and included. Needless to say, debugging situations like these is unpleasant.
>
> Prefixing all inclusions with the project name also makes sure that headers with common names (for example, `utility.hpp`) can coexist (for example, when installed into a system-wide directory, such as `/usr/include`). The prefix also plays an important role in supporting auto-generated headers.
>
> Note also that this header inclusion scheme is consistent with the module importation, for example:
>
> ```
> import hello.utility;
> ```
>
> Finally, note that while adding the project prefix to the `""` style inclusion (for example, `"libhello/hello.hpp"`) will make finding an unrelated header unlikely, there is still a possibility. And it is not clear why take the chance when there are no benefits (one such cited benefit is the ability to distinguish inclusions of "this project's headers" vs "external headers").

If you have to disregard every rule and recommendation in these guidelines but one, for example, because you are working on an existing library, then insist on this: **public header inclusions must use the library name as a directory prefix**.

The project's source subdirectory can have subdirectories of its own, for example, to organize the code into components. Naturally, header inclusions will need to contain such subdirectories, for example `<libhello/core/hello.hpp>`. When the project's headers are installed, this subdirectory hierarchy should be preserved.

If you need to separate public API headers/modules from implementation details, the convention is to place them into the `details/` subdirectory. For example:

```
libhello/
└── libhello/
    ├── details/
    │   └── utility.hpp
    └── ...
```

> If a project has truly private headers (for example, proprietary code) that must be clearly separated from public and implementation detail headers, then they can be placed into the `private/` subdirectory, next to `details/`. In a sense, this arrangement mimics the C++ public/protected/private member access.

If a project belongs to a *family of libraries* with a common name prefix, then it may use a nested source directory layout with a common top-level directory. As an example, let's say we have the `libstud-path` and `libstud-url` libraries that belong to the same `libstud` family. Their source subdirectory layouts could look like this:

```
libstud-path/
└── libstud/
    └── path/
        ├── path.hpp
        ├── path-io.hpp
        └── ...

libstud-url/
└── libstud/
    └── url/
        ├── url.hpp
        ├── url-io.hpp
        └── ...
```

With the header inclusion paths adjusted accordingly:

```
#include <libstud/path/path.hpp>
#include <libstud/url/url.hpp>
```

5 Source Naming

When naming source files, only use ASCII alphabetic characters, digits, as well as `_` (underscore) and `-` (minus). Use `.` (dot) only for extensions, that is, trailing parts of the name that *classify* the project's files. Examples of good names:

```
SmallVector.hpp
small-vector.hpp
small_vector.hpp
small-vector.test.cpp
```

Examples of bad names:

```
small+vector.hpp
small.vector.hpp
```

> If you are using `_` or `-` as word separators in filesystem names, pick one and use it consistently throughout the project.

The C source file extensions should be `.h` / `.c` and the C++ source file extension scheme should be `.?pp` :

```
file        .?pp

header      .hpp
module      .mpp
inline      .ipp
template    .tpp
source      .cpp
```

The use of inline and template files is optional. If used, they are included at the end of the header/module files and contain definitions of inline and non-inline template functions, respectively. The `.?pp` files with the same name are assumed to be related and are collectively called a *module*. [Note: This term is meant to correspond directly to a C++ module.]

> There are several reasons not to "reuse" the `.h` C header extension for C++ files:
>
> - There can be a need for both C and C++ headers for the same module.
> - It allows tools to accurately determine the language from the file name.
> - It is easier to search for C++ source code using wildcard patterns (`*.?pp`).
>
> The last two reasons are also why headers without extensions are probably not worth the trouble.

Source files corresponding to C++ modules need to embed a sufficient amount of module name suffix in their names to unambiguously resolve all the modules used in a project. When deriving file names from C++ module names, `.` (dot) should be replaced with either `_` (underscore), `-` (minus), a case change, or a directory separator, according to the project's file naming scheme. For example, if `libhello` had two modules, `hello.core` and `hello.extra`, then their interface units could be named as follows:

```
hello-core.mpp
hello-extra.mpp

hello_core.mpp
hello_extra.mpp

HelloCore.mpp
HelloExtra.mpp

hello/core.mpp
hello/extra.mpp
```

```
core.mpp
extra.mpp
```

As discussed in the next section, public module names should start with the project name and for such modules it is customary to omit this first component from file names (the last variant in the above example).

> See Building Modules in the `build2` documentation for a description of a module name to file name mapping algorithm that uses this naming scheme.

6 Source Contents

All macros defined by a project, such as include guards, version and symbol export macros, etc., must all start with the project name (including the `lib` prefix for libraries), for example `LIBHELLO_VERSION`. Similarly, the library's namespace and module names (both public and implementation detail) should all start with the library name but without the `lib` prefix. For example:

```
// libhello/hello.mpp

export module hello.core;

namespace hello
{
  ...
}
```

An executable project may use a namespace (in which case it is natural to call it after the project) and its (private) modules shouldn't be qualified with the project name (in order not to clash with similarly named modules from the corresponding library, if any). [Note: A library may also have private modules in which case they shouldn't be qualified either.]

> Hopefully by now the recommendation for the `lib` prefix should be easy to understand: oftentimes executables and libraries come in pairs, for example `hello` and `libhello`, with the reusable functionality being factored out from the executable into the library. It is natural to want to use the same name *stem* (`hello` in our case) for both.
>
> The above naming scheme (with the `lib` prefix present in some names but not others) is carefully chosen to allow such library/executable pairs to coexist and be used together without too much friction. For example, both the library and executable can have a header called `utility.hpp` with the executable being able to include both and even get the "merged" functionality without extra effort (since they use the same namespace):
>
> ```
> // hello/hello.cpp
>
> #include <hello/utility.hpp>
> #include <libhello/utility.hpp>
>
> namespace hello
> {
>   // Contains names from both utilities.
> }
> ```

7 Tests

A project may have *unit* and/or *functional/integration* tests. Unit tests exercise each module's (potentially private) functionality in isolation. In contrast, functional/integration tests exercise the project via its public API, just like the real users of the project would.

7.1 Unit Tests

A source file that implements a module's unit tests should be placed next to that module's files and be called with the module's name plus the `.test` second-level extension. For example:

```
libhello/
└── libhello/
    ├── hello.hpp
    ├── hello.cpp
    └── hello.test.cpp
```

All source files (that is, headers, modules, etc) with the `.test` second-level extension are assumed to belong to unit tests and should be excluded from the library/executable build.

> The use of a second-level extension rather than a file name suffix (for example, `hello-test.cpp`) has several advantages. Firstly, it is conceptually fitting since being a test is just another *axis of classification*, not unlike being a C++ source file.
>
> Also, `.test` makes things more robust at the build system level where files belonging to unit tests can be handled with a wildcard (`*.test.cpp`). In contrast, `*-test.cpp` is more likely to pick up something that fits the naming schema accidentally, for example, `midterm-test.cpp`.

Each unit test source file should implement a standalone executable (that is, define `main()`). It should be possible to run such an executable without any command line arguments or inputs in order to perform all the unit tests of a module. Such an executable should indicate success by terminating normally with the zero exit code.

7.2 Functional/Integration Tests

A project's functional/integration tests should go into the `tests/` subdirectory. Each such test should reside in a separate subdirectory, potentially organized into nested subdirectories (for instance, to correspond to the source directory components). For example, if we were creating an XML parsing and serialization library, then its `tests/` could have the following layout:

```
tests/
├── basics/
```

```
│   └── driver.cpp
├── parser/
│   ├── pull/
│   │   └── driver.cpp
│   └── push/
│       └── driver.cpp
└── serializer/
    └── ...
```

> A reasonable question to ask is why place functional/integration tests into a separate subdirectory while unit tests next to the modules that they test. The main reason is the ability to run the former against an installed version of a library (if you think about it, the way most build systems do it currently is backwards: they first run the tests and then install the result, which may end up with missing headers, wrong rpaths, etc).
>
> So some build systems (such as `build2`) have a notion of subprojects that can be configured and built independently of their superproject. But to make something a subproject, it normally has to be a separate directory.

Each functional/integration test should result in a single executable that indicates success by terminating normally with the zero exit code. While such tests may require command line arguments, inputs, and/or, output analysis, to facilitate packaging, such requirements should be avoided, if possible.

8 Acknowledgments

Thanks to Titus Winters, Robert Schumacher, and Joël Lamotte for providing valuable feedback on earlier drafts of this document.