How to organize the project tree for a C++ project using nmake?

Asked 16 years, 1 month ago Modified 10 years, 9 months ago

Viewed 4k times

17

There seems to be two major conventions for organizing project files and then many variations.

Convention 1: High-level type directories, project sub-directories

For example, the wxWidgets project uses this style:

```
/solution
   /bin
      /prj1
      /prj2
   /include
      /prj1
      /prj2
   /lib
      /prj1
      /prj2
   /src
      /prj1
      /prj2
   /test
      /prj1
      /prj2
```

Pros:

- If there are project dependencies, they can be managed from a single file
- Flat build file structure

Cons:

- Since test has its own header and cpp files, when you generate the unit test applications for EXE files rather than libraries, they need to include the object files from the application you are testing. This requires you to create inference rules and expand out relative paths for all the source files.
- Reusing any of the projects in another solution requires you to extract the proper files out of the tree structure and modify any build scripts

Convention 2: High-level project directories, type sub-directories

For example, the Wireshark project uses this style

```
/solution
   /prj1
      /bin
      /include
      /lib
      /src
      /test
   /prj2
      /bin
      /include
      /lib
      /src
      /test
```

Pros:

- Projects themselves are self-contained within their folders, making them easier to move and reuse
- Allows for shorter inference rules in the build tools
- Facilitates hierarchical build scripts

Cons:

- If there are dependencies between projects, you need an additional layer of build scripts above the project directories to manage the build order

We are currently using convention 1 on our project and so far it has worked fairly well. Now, I am in the process of adding unit testing (via CxxTest) and facilitating the migration to continuous integration using nmake, convention 1 is causing some serious headaches in the creation of the proper nmake files.

My primary requirements/goals are:

- Reduce the level of effort to maintain the build scripts of the entire solution.
- De-couple projects and their build steps within a solution from other projects.
- Facilitate continuous integration via the use of build scripts for check-out to release media generation for each commit (obviously leveraging other tools such as CruiseControl as well).

- Make adding or removing additional projects or source files as easy and least error-prone as possible for the developers.

So I ask:

- Are there other pros and cons of either of these methods?
- Is there a clear agrument that favors only one of these conventions?

build-automation project-organization nmake

Share

Improve this question

Follow

edited Mar 28, 2014 at 2:30

Peter Mortensen
31.6k ● 22 ● 109 ● 133

asked Nov 5, 2008 at 18:56

Zach Burlingame
13.8k ● 15 ● 58 ● 65

I deleted my original answer about using Lakos's Book, since it does not actually answer the question. I'd still recommend the book though: Large Scale C++ Software Design. – grieve Nov 5, 2008 at 21:44

Thank you, it was already on my reading wishlist on Amazon =] – Zach Burlingame Nov 5, 2008 at 23:48

2 Answers

Sorted by: Highest score (default)

4

[A partial answer.]

In "Convention 2: High-level project dirs, type sub-directories," your single con is

> If there are dependencies between projects, you need an additional layer of build scripts above the project directories to manage the build order

That can also be viewed as a pro, in many projects.

If you have a lot of repetitive general definitions, one would probably want an include file for the build scripts, where solution-wide constants & parameters could be defined. So the "additional layer of build scripts" will frequently happen anyway, even if there are no (direct) dependencies.

It's a pro in that there's still room for a more modular approach in building. On the other hand, if you want to reuse a project in another, unrelated solution, you would need to compose a different definitions file. (On the other other hand, if there were a single build file for the whole solution, as in Convention 1, you would need a different build script.) As for your maintenance requirement, that's (IMO) very project-dependent.

My feeling leans towards Convention 2, but it's far from a clear win. In fact, your experience with Convention 1, which was working well until recently, may be the biggest pro of all: a team of people with experience with a certain organization is a valuable asset.

Share Improve this answer Follow

answered Nov 6, 2008 at 7:53
JXG
7,403 ● 8 ● 36 ● 63

I completely agree that that Con can in-fact be a Pro. I was struggling to come up with cons for it and some of that is mostly likely due to the fact that I'm simply more familiar with Convention 1. – Zach Burlingame Nov 6, 2008 at 14:17

1

Consider using NTFS junction points so you can have both organizations at once. Quick definition: "a junction point is Microsoft's implementation of symbolic links but it only works for directories."

Use Convention 2 for the "real" layout, because it makes the projects easy to move around. Then make a Convention 1 view:

```
mkdir /solution/test
linkd /solution/test/prj1 /solution/prj1/test
linkd /solution/test/prj2 /solution/prj2/test
```

Now you have ...

```
/solution
  /test
```

```
    /prj1
    /prj2
```

... which was the desired result.

You could do the same thing for /src or the other directories if you find it beneficial. Test scripts that benefit from a Convention 1 view live in /solution/test.

Share Improve this answer

Follow

answered Nov 6, 2008 at 12:23

Thomas L Holaday
13.8k ● 6 ● 43 ● 52

1 An interesting idea, but it poses a couple of issues. For one, properly maintaining the "real" and "linked" trees is undesirable. More important, SVN (and most Windows applications, including Explorer.exe) don't handle hardlinks or junctions properly and can lead to undesirable command results. – Zach Burlingame Nov 9, 2008 at 2:35

1 It's unfortunate that junction points (symlinks) just aren't part of the Windows "culture" the way they are on *nix, since they do simplify certain tasks quite a bit. Hopefully application and tool support for them will improve with time.
– j_random_hacker May 25, 2009 at 3:44