Organising source code

We've all seen it, one directory containing lots and lots of files. Where to start? Where is main? How do they fit together? This is the classic *Topsy system*, someone ran the wizard once and just kept on adding more and more files.

While it is simple to put all our files in one place we are missing an opportunity to communicate information about the system. The original designer may know these five files form a discrete group, and another six over there form another module but this isn't obvious, that information has been lost, recovering it takes time.

Consequently our understanding slowed, and changes to the system are delayed – the software is resisting change. We need to split the system in to comprehensible, logical, cohesive modules; we can then use the directory tree structure to convey the system structure.

Dividing our system across several directories has other advantages. It becomes easier for multiple developers to work on the system at the same time, and it becomes easier to transfer modules between multiple projects.

The directory structure of a project is closely linked with the source code control system employed – one mirrors the other. We cannot consider the layout of files without talking about the source code control too - once we commit files to source code control it becomes more difficult to move them to other directories or rename them.

Libraries, files, and directories represent the most physical manifestation of source code short of a printout. Consequently they form a key part of our overall strategy. Neglecting these aspects leads to blob like software that lacks cohesion.

Splitting the system into modules

Many programming courses start by teaching their students the need to divide systems into modules. *Modularization* has moved beyond buzz-word status, we take for granted that it is *good thing* and all systems should exhibit it. Some of the benefits cited usually include:

? *Comprehensibility*: while modularization helps us understand smaller elements of a system we need to be able to integrate this knowledge. Integrative knowledge is more difficult to express.

? *Division of labour*: if is easier for multiple developers to work on a modularised system then a monolithic one. We also get *specialisation of labour* where experts in one aspect of the system can work on one module, and other experts on other modules.

? Modularization should help focus our minds on cohesion, coupling, dependencies, division of tasks and such, this should all make the system easier to change.

Reuse is often cited as another advantage of modularization, given the current debate on reuse I won't cite it as an automatic benefit. However, if we wish to share elements between projects then there must be some division of the source code.

But what are our modules? *Module* and *modularization* are such overloaded words we're never really sure what they mean. *Component* has similar problems.

Individual files can be a module, but that is too fine grained for most purposes. And if a module is a file why use the word module? And what difference does it make?

One of our usual objectives in defining modules is that we wish to practice *information hiding* it seems to me that the correct level to define our modules is the level at which we can actively hide some information, that is, hide some implementation.

Once our C or C++ code is compiled to object code we can hide the implementation since we only need distribute the header files and the object file. Still, the object file has a one-to-one relationship with the source file so we're not hiding much.

We need a bigger unit to hide in. When we bundle many object files together we get a static library. This is more promising. Our code can interface to the library by including one or more header files and we shouldn't need to care whether the library is made up of one file, two files, or 25.

Static libraries are simple to create and use. Once compilation and linking are complete then static libraries presents no additional overheads or requirements, we can have as many of them as we want at no additional run-time cost. Hence, they are well suited to be are building blocks when decomposing a system into discrete chunks.

Dynamic link libraries are more complicated, by no means are they simply "static libraries which have been linked differently." We must consider run-time issues, where are the libraries found? How do we find them? Do we have the right version? Dynamic libraries have their place but they should not be the basic building block.

When we create a static library we want to hide a secret inside the library. The secret is *implementation detail*. We want the library to represent an idea, and we want to hide the realisation of the idea from the rest of the system. To this end, the library needs to be highly cohesive, that is, it needs to express its ideas fully but no more than need be, it should have lots of bells-and-whistles. The library also needs to pay attention to what it depends on, how connected it is to other modules in the system, that is, we want to minimise coupling.

We can't reason about the cohesion and coupling of every file, class and function in a system, that would take forever. While individual developers may consider these forces within the library module at a system level we would be overwhelmed by such details.

Static library modules represent the basic ideas from which a system is built. Since each one contains a complete idea we should expect to have many static libraries in our system. Many is good, it shows that your ideas are discrete and can be expressed individually.

Although you can cram more than one idea into a library you usually know when you are doing so. It is pretty obvious when the library is called "Logging" and you are putting database update function in that something is wrong.

It is also possible to fracture and idea and split it across multiple libraries, but again it is pretty obvious. You quickly notice that library "Logging" always requires library "LogMessage" and something isn't quite right here.

Good systems are decomposed into many distinct static libraries - we should prepare for and encourage this. On top of the libraries we will find at least one application which results in an executable program. It may only comprise one file, a main.cpp, with the bulk of the code farmed out to static libraries.

You may well find that your project produces several applications, when this happens you can benefit from good modularization. There is no need for each application to provide its own logging system, you use the logging library.

Is this reuse? Well, that depends on your definition of reuse. I would argue that you are producing a family of programs with common characteristics for which you use common code. In time you may transfer some of this code to other projects.

How do we encourage modularization? Well, we start by providing a structure into which we can modularization our project. Since we will be writing files, we need a directory tree to place them in.

The Directory tree

In the early days of a project we may like to work light, especially if there is just one developer on the project. But very quickly a project crosses a line, usually when a second developer starts work, or you decide that you could pull in code from a previous project. Once you've crossed this line you need to structure the work area of the project, that is, the directory layout.

Obviously we want a logical directory structure but we also want one we can add to. We need to be able to create sub-directories for new modules, and we don't want to get overwhelmed by directories. It is better to have many small modules, in many directories, than several "catch all" modules in a few directories. Above all, we need to give ourselves space.

We want to use the directory tree to partition the system into recognisable chunks: all the database files in one directory, all the logging files in another, and so on. When someone comes new to the system each chunk is clearly defined. Of course, they still have to understand the insides of each chunk, and how they fit together but you are not faced with one overwhelming mass of files.

The directory structure we use should map directly into the structure used in our source code control system. The two are closely intertwined, and although I've tried to separate the rational for each I can't, there is one hierarchy and it should apply in the directory tree and in the source code control tree.

Some control systems allow to break this link and check files out to different locations. On occasions this can be useful but if you find you need to do this regularly you should consider why. Your structure is lacking something. Even with the best intentions breaking the link becomes troublesome, in the long run it is better to come up with a solution which does break the link between directory hierarchy and source code hierarchy.

Where is the root?

All trees need to be routed somewhere and software trees are no different. If we always refer to our directories by relative paths we need not care where they are rooted. However, experience shows that this eventually breaks down, at sometime we need to refer to the absolute location of files. It is a lot easier to reason about a path like /home/allan/develop/lib/include/logging than it reason about ../../../include/logging – try shouting the latter across the room.

In the Unix world there is only one ultimate root, /, but we all have local roots, e.g. /home/allan. Usually our trees are rooted in our home directory but not always. Typically we set an environment variable to the point where our tree is rooted and append from there, so we get PROJECT_ROOT=/home/allan/develop, and $PROJECT_ROOT/lib/include/logging.

(I was confused for far too long over Unix environment variables, sometimes they just wouldn't work for me. What I was failing to appreciate is that there are two types: shell variables, and exported variables.)

In the one-root-per-disc world of Microsoft things are a little more complex. Traditionally I would use the `subst` command to create a virtual drive on my machine, this I could point wherever I liked – it is worth putting this command in a start up script.

```
subst w: d:\develop
```

Thus, each developer can have their directory tree where they like, their C: drive, or D:, at the root, or within another tree.

More recently I've moved over to the Unix way of doing things even in the Microsoft world. As it happens .dsp project files are happy to pick up environment variables so you can use the same technique as in Unix.

Unfortunately, Microsoft has made environment variables a lot more hassle under Windows than Unix. Unix is simple: set them in your shell .rc file and change them at the command line. The .rc file can be copied, e-mailed and edited easily. Under Windows, you need to go fiddling in the control panel, and the location seems to move slightly with each version of Windows.

This may seem like a lot of unnecessary work but it pays for itself if you ever need to maintain two different development tree on the same machine, say a maintenance version 2.1.x and the new development 3.0.

The External Tree

It is it increasingly unusual to find a system that doesn't use any third party code or libraries. Whether these are commercial libraries like RogueWave, Open Source projects like Xerces or future standards like Boost we are increasingly dependent on code we have not created.

It is important to differentiate between in-house code, which we have created and own the rights to, and code which is external to our organisation. The development cycle for these two sources of code is very different. Our own code is changing according to a cycle which we dictate, the third party code changes whether we want it to or not, of course, we decide if and when we accept these changes so in the meantime the code is static.

Once we've decided on our root, and how we refer to it, we need to split our directory tree to show what is ours and what's not. Typically this means we create an *External* directory tree which contains third party code, e.g. `/home/allan/develop/external`, while our own code goes in a separate tree such as `/home/allan/develop/ProjectFire`. (Sometimes the External tree is called "third party" tree, but this gets confusion, is it: 3rdParty, or ThirdParty, or 3Party, where are the capitals?)

Sometime we find one external tree can be used for several projects. This may mean we have multiple versions of the same product in the tree, say Xerces 1.6 and Xerces 1.7. We have two solutions here: one if to include everything in one external tree and reference what we need, the second is multiple external trees, we may have /home/allan/develop/external/v2.1 and /home/allan/develop/external/v3.0. Which way you jump depends on your project requirements.

It is not normally a good idea to try and delta one version of an external product on top of a previous version. The complications of new files, removed files and renamed files usually make this a lot of effort for little reward. It is simpler to just allow multiple versions in the tree.

On my current project I have environment variables for almost everything, so I have one external tree and within that I can point XERCES_ROOT to the version of Xerces I need.

The other half of the development tree is your own source code. You'll probably find this is dwarfed by the third party code, I normally find that even on my biggest projects a Zip file of our code will happily fit on a floppy disc but the third party stuff needs it's own CD.

Of course, life isn't quiet this straightforward, you may well have code from elsewhere in your organisation, perhaps this is supplied by the *infrastructure team*, or the *client application group* or the New York office. Sometime you'll want to share the tree with them, sometimes you'll want to treat them as external code, or maybe you split you tree three ways: external, enterprise and team. It depends on how much you need to isolate yourself from these developers.

Align libraries with namespaces

If your following my guidelines from above the chances are you've got at least one executable application, several DLLs and lots of static libraries. Each application, DLL and library needs its own space – that is: its own directory. Only by giving them their own directories can you explicitly spell out what belongs where.

Where static libraries are concerned you need to split the files into two directories. Most, of the code in a static library is contained in .cpp files (or .c, or .cxx or whatever.) This is implementation detail. You want to put this out of the way – out of sight, out of mind. However, the interface to the library is going to be expose in a set of .h files, these need to be put somewhere publicly visible, somewhere obvious, somewhere well know.

Traditionally we do this by creating a lib subdirectory, inside this we would have:

```
/lib

      /include

      /Logging

      /AccessControl
```

(Although most OS's now allow you to have spaces in directory and filenames they are still best avoided, they complicate matters.)

In our `lib/Logging` directory we would put all files implementing our logging system, the files exposing the public interface would be put in the `lib/include` directory where we can all find them. Similarly, lib`/AccessControl` contains the implementation for our access system, and the public interface files are also put in `lib/include`.

Using one include directory we quickly fill it with a lot of disparate files which adds nothing to our structural information. We could leave them with the implementation files – but now we can't tell what is a private, implementation only header file, and what is a public interface file.

Alternatively, we could put them all in separate directories but this could mean we end up with lots and lots of –I options on the link line, imaging:

```
gcc   –I $PROJECT_ROOT/libs/include/Logging

      -I $PROJECT_ROOT/libs/include/AccessControl ....
```

Well who cares what it looks like? Does it really matter? No, but, each time you add a new library you need to change your makefile to specify the new include directory.

A better solution is to specify one root include directory, and within our code specify the actual library where interested in, hence we get

```
gcc  -I $PROJECT_ROOT/libs/include
```

and

```
// main.cpp

#include “Logging/LogManager.hpp”

#include “AccessControl/AccessOptions.hpp”
```

The real power of this comes when we align with namespaces. So, each library has its own namespace and the namespace corresponds to the sub-directory. Continuing the above example we get:

```
...

int main(int argc, char* argv[]) {

     Logging::Logger log;

     AccessControl::User user(argv[1]);

     ...
```

Looking back to the source tree, we should also think about balancing it is little bit, it now seems a little lob-sided, so we get:

```
/lib

     /include

            /AcessControl

            /Logging

            /Utils

     /source

            /AcessControl

            /Logging

            /Utils
```

The extra level of directories may seem surplus but actually helps space the project quite well. When you put this altogether you get a very powerful technique for managing your files and module structure, it really pays off.

A couple of points to note however: firstly, not all header files will go in the `/include` directories. Some don’t represent an interface to the library, they are not intended for public use so they should only exist in the `/source` directories. It is good to make a clear distinction between what is available to the general public and what is considered local implementation detail.

Second, I’ve taken to prefixing my header guards with the namespace name as well, so:

```
#ifndef ACCESSMGR_HPP
```

Becomes:

```
        #ifndef ACCESSCONTROL_ACCESSMGR_HPP
```

Once your free of thinking up completely unique filenames you quickly find several `Factory.hpp` files appearing, it doesn't really matter because one will be `Logging/Factory` and one will be `AccessControl/Factory`. However, some debuggers (OK, I'm thinking of one from Redmond) can have problems telling the files apart.

If you already think of namespaces as program modules this technique may be obvious, if your still thinking of namespaces as a convenient way to avoid name clashes then you need haven't realised their full potential yet. A namespace is a C++ language module; a static library is the natural corollary.

Lastly, and fairly obviously, this doesn't quite apply to template files where the implementation must be exposed in the same file as the interface. I'll still tend to stick with my solution and place them in the `lib/include/xxx` directory because nowhere else really makes sense. Hopefully, in time, the C++ compiler vendors will resolve this problem, but in the meantime systems where the majority of the code is templates are fairly rare.

Dlls

Dll's should be treated just like static libraries, that is, I give them a Dlls' directory and split this into source and includes. Dll's are different from static libraries, the linkage rules are different they are used for different reasons – some people regard use of one over the other as fairly arbitrary but with experience you come to see them as two very different beasts.

One of my personal rules is to avoid subtlety in design. Let's call it Kelly's law of Software Subtlety:

Subtlety is bad If it is different make it obvious - Write it BIG

Since DLLs are different to libraries put them somewhere else.

As your system grows you can easily find you have 10 or more static libraries in to manage and perhaps another 3 or 4 DLLs. If we separate out Dll's, we give ourselves more space. There is a natural difference here so use it.

We need to add another –I to our compile line, but it is just one more, the same namespace alignment scheme can be used for DLLs provided you don't need to specify C linkage for any of your functions.

Applications

Even if your objective is to produce just one final application, say, server.exe the chances are you will end up with more than one executable, even if all but one are trivial. Executables are the top of the food chain when it comes to source code so it pays to put them there - straight off the project root.

If however, your project is going to produce many application you may want to avoid littering the project root with lots of directories. In this case create a `project_root/app` directory and create a sub-directory for each on there.

There is no need for to split header files from source files because the application does expose it's internals like this. If you find you need to access application files, and you see things like `#include "../otherapp/widget.hpp"` appearing in your code it is an indication that there is some shared functionality that belongs in a library where it is readily accessible to all. The directory tree is highlighting a refactoring needed in your system.

One some projects you may find that one or more applications are large enough to warrant being broken into libraries by themselves. If so then don't hesitate, go for it!

Each one becomes a mini-project in it's own right, apply the tree design I've just outlined to each application. You'll probably find some common libraries shared between them all, they stay where they are, but for a large application it should have its own library structure. This is just recursion, apply the pattern over again taking the application's directory as your new root.

Putting it together

You'll probably find that you have other stuff which needs to be in the tree, makefiles, documentation and such. Where these belong to a component, say documentation for your logging library then place the documents same directory as the library. Where they are common, say system documentation, place them in a docs directory from the root – if necessary divide it.

In the case of makefiles you'll find that some are common and need to be placed in a well known place in the tree, but most are specific to individual elements – indeed each library, dll, application, should have its own.

Your source tree should not be looking something like this:

```
$project_root
     /apps
            /repairtool
            /server
            /testtool
     /dlls
            /include
                   /BlueWidgets
                   /RedWidgets
            /source
                   /BlueWidgets
                   /RedWidget
     /docs
     /libs
            /include
                   /AccessControl
                   /Logging
                   /Utils
            /source
                   /AccessControl
                   /Logging
                   /Utils
     /make
```

At first sight this may seem a bit excessive, but the important point is it gives you space, it gives you organisation, you are free to do what you like in any sub-directory and it won't interfere with another. This model will scale, we are building in-depth for extendibility.

External tree and source control

Nor do I have a good answer for the question "Do we check in the external tree?" There are three possible answers here:

? Check nothing in: the code has come from else where, you can always get it again, download it, install from CD. Maybe you want to burn some CDs with downloads on so you can always get old versions. Of course, what happens when you fix a bug in external code? I once found a fault in the ACE library, I devised a fix and contributed it back but it was several months before a version of ACE was released with my fix. In the meantime we checked-in my fix to our ACE copy in CVS and carried on working.

? Check in source code: if you have the source code you can recreate the binaries, this will save people having to locate several dozen different resources. And it helps when you make a change to external code. However, each developer needs their own copies of the binaries - access over a network to a common store can substantially slow compile time - but it can be quite time consuming to build lots of third party libraries.

? Check in source and binaries: source control systems aren't really built for binaries and they rapidly bloat when you check in everything, and if we do then developers need even bigger hard discs to get trees containing lots of stuff they don't actually want.

Nor does the external tree contain all the third party products in your system. Do you check in your compiler? Perl? Your OS?

Source control is not confined to your source code control system, it should encompass all the elements needed by your system: compilers, OS, patches, etc, etc. It is unrealistic to put everything under source control so you need to look elsewhere.

The best solution I know to this dilemma is:

1) Only check in source code, this means you can tinker with Boost, ACE, or whatever is you need to.

2) Build the binaries under clean conditions – as you would your own source but place it somewhere generally available, say a Samba drive. This is a kind of cache and ensures that everyone is using the same thing and saves everyone rebuilding it. Some people may care to place some of this stuff on their local hard disc.

3) Burn CD copies of all third party product used to build your system, be they libraries, compilers or whatever, and put them in a safe, well known location.

The objective is: to be able to take a new machine and only use what is in the cupboard and source code control be able to build your entire system.

Conclusion

Our directory structure effects our source code. At the simplest level this is shown in our makefiles, makefiles are the glue that links the two, makefiles explain how to integrate the system. Directory trees always have an effect on code, rather than hide this detail we should use it to our advantage. The directory tree can encode important details about the system structure and organisation, and its extension points.

Much comes down to your work environment, your schedule, your team but these issues are important. The ideas I've laid out come up again and again, each team will have slightly different needs but all solutions exhibit a general similarity.

Above all you need to actively organise your source code, external tools and resources. These things don't just happen. This organisation is an integral part of your software development strategy and helps communication.