Build systems

In my last essay I discussed the structure of the directory tree containing source code, here I want to discuss the systems we use to build source code, the directory tree structure plays an important part in this. The structure we use to hold our source code, and the mechanism we use to convert that source code into programs are all part of the same problem.

Source code is the representation of everything you know about the problem and solution domain, creating it is a process of blood, sweat and tears, but unless you can turn it into executable programs it is worthless. Before it can generate revenue it must be built.

First and foremost we must be able to build our source code. This may seem obvious but there is nothing more depressing than being given source code and finding we can't build it. The .dsp won't load, or make bombs out immediately – or there just isn't a make file.

It helps to ask two questions:

? Given a new PC how long does it take to get it building our source?

? If we hire a new developer how difficult is it for them to build the system?

One company I encountered considered it a *right of passage* for a new developer to build the server, this is quite adversarial and depressing to a new developer.

Objectives

Let's consider a few objectives for our build system:

? Reliable: no point in having a system that is slightly random.

? Repeatable: having a system that only works three out of four weeks is no good.

? Understandable: build systems tend to become increasing cryptic over time, especially if you use some of the more obscure syntax of make.

? Fault aware: things will go wrong with our build system, we don't expect fault tolerance but we can aim for fault awareness. When a build fails we would like as much information on why and how as possible.

There are more objectives we may like to add over time: multi-platform capable, automated, multiple build types (e.g. debug, release), fast and so on but lets start with a short list.

We need to think of the build system as part of the source code. Makefiles are as much part of your system as any other source code file, be it a .cpp, or .hpp. They should be subject to the same source code control as any other file. Your makefiles are essential to your applications.

And everyone should use the same build process, and the same makefiles. It is counter productive if you using one set of makefiles and Jo using another set. What if you set the /GX options and he sets the /O2 options? Subtle differences will emerge, subtle faults will appear – they will even appear and disappear at "random" intervals.

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

Unfortunately this leads to a contradiction in the build system. Developers want a system that is easy to use, integrates with our tools, is fast, does the minimal rebuild possible. However, the build master wants a 100% repeatable process – no typing make again if it fails the first time – and their definition of "easy to use" is different from ours.

Where we have a overnight batch build we have slightly different objectives again: speed is less of an issue, but automation is paramount.

Some of these difference can be easily resolved (e.g. have the build master use the same build process as the overnight build, maybe even take a release candidate from the overnight build), other difference aren't reconcilable and compromise is needed.

The clean build

When developing a build system my first milestone is:

? Given a clean machine, I should be able to install a set of packages from a given list, get files from source code control, and perform a build.

If we can't do this, do we have any hope of ever building our software? Consider the alternatives. Many of us have been there, you start work at a new company and try and build the code. It fails, someone says "Yes, you need to install Python", you install Python. It fails. Then the someone says "Yes, you have to install a version 2.1." Next time it fails: "Can't find \boost\any.hpp", after much searching you discover that everyone else has BOOST_ROOT set in the environment but has forgotten about it. And so on.

Once I can rebuild on a clean machine look to repeatability:

? Automate the build to happen at 1am every night.

(Does 1am sound OK to you? Or are your developers frequently working until 2am? Are you really sure you want them checking in when they are bleary eyed? I'm told that Apple used to have a rule against check-ins later than 10pm. Perhaps more problematic than insomniac developers is what to do when your development spans multiple time zones, or when the build takes a long, long time.)

If you can't automate your build how do you know it is repeatable? How do you know it doesn't rely on Pete blowing the tree away everyday at 3pm? Even if you document your build process ("install Visual C++, download Boost, build the utils library, build the application") how do you know the document is accurate and up to date? The process should be the documentation.

Wile creating repeatable build you need to strive to make the build aware of potential problems. Have it issue meaningful message if a package is missing, have it log output to a file, even comment the makefiles!

With these elements in place we are half way to meeting the objectives outlined above.

Environment variables

Variables, as always, are key to capturing variance and the same is true with environment variables. In the build system we use them to express configuration details (where to find the external tree), options (build debug or release), set defaults (developers will default to debug builds, build master will default to release and the overnight build should do both) and communicate between the makefiles.

Unix folks have always been friendly with environment variables while Windows people, on the other hand, see environment variables as a hang over from the DOS day – who needs them when you have the registry? However, they are just as important on Windows machines as Unix machine for two reasons. First, they communicate with the compiler and make much better than the registry, and second, they provide a common mechanism so scripts can be used on Unix too.

You can use an environment variable like PROJECT_ROOT almost as well in Windows as in Unix. The project configuration in the Microsoft IDE does its best to confuse you, the "project settings" dialog is even more fiddly than the control panel and uses the $(PROJECT_ROOT) syntax like make and bash, and unlike the Windows command line %PROJECT_ROOT% syntax.

Here is a section of my current .bashrc file:

```
export PROJECT_ROOT=f:/xml
export EXTERNAL_ROOT=f:/external/lib
export BOOST_ROOT=f:/external/boost_1_27_0
export GSOAP_ROOT=$EXTERNAL_ROOT/soapcpp-win32-2.1.3
export BERKELEYDB_ROOT=$EXTERNAL_ROOT/db-4.1.15
```

I'm running the Cygwin toolkit for Windows which provides a Unix-like environment. This gives me a rich shell environment – far superior to the command shell provided by Microsoft. It is also, give or take a drive reference, compatible with the bash shell on my Linux box. Even when I'm not engaged in cross-platform work I run a Unix shell, currently Cygwin bash but MKS also do a good Korn shell.

Returning to the .bashrc section, I'm mostly defining variables for third party tools and libraries in terms of previous variables. Not always, because (a) I'm a little bit random myself, (b) I sometimes try

new versions of things and they may live in a different place until I decide what to do with them. The important thing is I have flexibility.

Now, take a look at a section from one of my make files:

```
ifeq ($(PROJECT_ROOT),)
$(error Error: PROJECT_ROOT not set)
endif

ifeq ($(EXTERNAL_ROOT),)
export EXTERNAL_ROOT=$(PROJECT_ROOT)/external
$(warning Warning: EXTERNAL_ROOT not set using default
$(EXTERNAL_ROOT))
endif

ifeq ($(BOOST_ROOT),)
export BOOST_ROOT=$(EXTERNAL_ROOT)/boost_1_26_0
$(warning Warning: BOOST_ROOT not set using default $(BOOST_ROOT))
endif
```

The makefile will check to see if it can find the important variables. If it can't find PROJECT_ROOT then all bets are off, give up now. If it can't find EXTERNAL_ROOT then make a guess, as it happens the guess it will make here is bad but as long as I've set in my environment it doesn't really matter. What is important is that the system is aware of a potential problem, it issues a warning but attempts to carry on.

Next it moves on to check a long list of third party sources. Again, if it can't find them it makes an educated guess and gives me a warning. In this way I don't need to set a large number of variables to get the build up and running, just a couple of key ones, although I have the option to do things differently, to put my external libraries somewhere else.

Importantly this is self documenting, if I need to know what third party packages I should download I can look at the makefile, the same file which will be used to find them when I do the build, the makefile *is* the documentation.

Secrets of Make

The original Unix make program must have the most cryptic syntaxes developed before the advent of Perl. Over the years it has put many developers off using it – and sent many running into the arms of Microsoft and their crippled make system using .dsw and .dsp files.

There are two reasons to ditch your Microsoft build system in favour of real make: first it is massively more flexible and powerful (although it is also easier to shoot yourself in the foot). Secondly: it allows you to construct true hierarchies, and perform recursive builds. In contrast Microsoft provide only one level of depth – that of grouping .dsp files in a .dsw.

There are several secrets about make you should be aware of before continuing:

? GNU make (sometimes called gmake) is a very different creature to the original Unix make, the old syntax is still there but there is new simpler syntax which makes it far easier to program.

? Rules and variables (confusingly called macros) are key to a good makefile.

? Rules specify targets.

? Variables can be picked up from your shell environment, set on the command line invoking make, or set within the make script.

? Rules can be specified in terns of environment variables. If the environment variables change, the rules change, this includes targets.

? By default, make will execute the first rule (target) it sees when processing a makescript. Unless another target is specified on the command line only the first rule will be processed, of course, the first rule may cause other rules to be invoked. (By convention this rule is usually called `all`.)

? The order of rules is not important - exceptions being the first rule, duplicate rules and includes.

? You can include other makefiles in your makefile.

? If you include a file that does not exist, make will turn in on itself to see if it knows a rule to create the file you want included before it executes the first rule.

? Except rules invoked to find or generate include files, no rules will be executed until all includes have been processed, then the first rule encountered will be processed.

? Make comes with a set of in built, default, rules which are useful for small programs but not much else. Many of these rules are legacy rules (e.g. compiling .pas files) and it is usually easier to disable them (with .SUFFIX) and specify your own.

? If things are too difficult to do in make you can always run a shell script from within make.

Everything I talk about here is specifically about GNU make. Your system may come with another make program, some of what I say will still be relevant but not all. Since there as almost as many make programs as their are compilers it is impossible to cover them all.

Before continuing let me point out that when I speak of *build systems*, I'm talking about the whole build process, source code extraction, build scripts, process and make files. When I talk of *make systems*, I'm talking specifically about the make scripts which run the compiler and other tools.

Make sandwiches

Make systems usually end up as a three layer sandwich. The first layer specifies the environment variables to be used, this sets the foundations of what is to come. The second layer is thin but essential: it specifies the first rule. The third layer specifies everything else, that is: the other rules needed.

We want our make system to be easy to use, ideally we should just type `make`, and everything will be built. Or rather, when we type `make`, we want everything in our current directory to be made, and if necessary, anything below the current directory.

If no file is specified on the command line make will attempt to process the file `Makefile` – there are several other names it will try before or after this, e.g. `GNUmakefile`, so consult your documentation.

Since the environment variables and rules are fixed it is this element of the sandwich that changes, the makefile in our immediate proximity is the filling in the sandwich, and it is here that execution begins, this is the interesting bit between two bits of bread. Most fillings will look very similar, they pull in the definitions, specify the first rule, and then pull in the other rules.

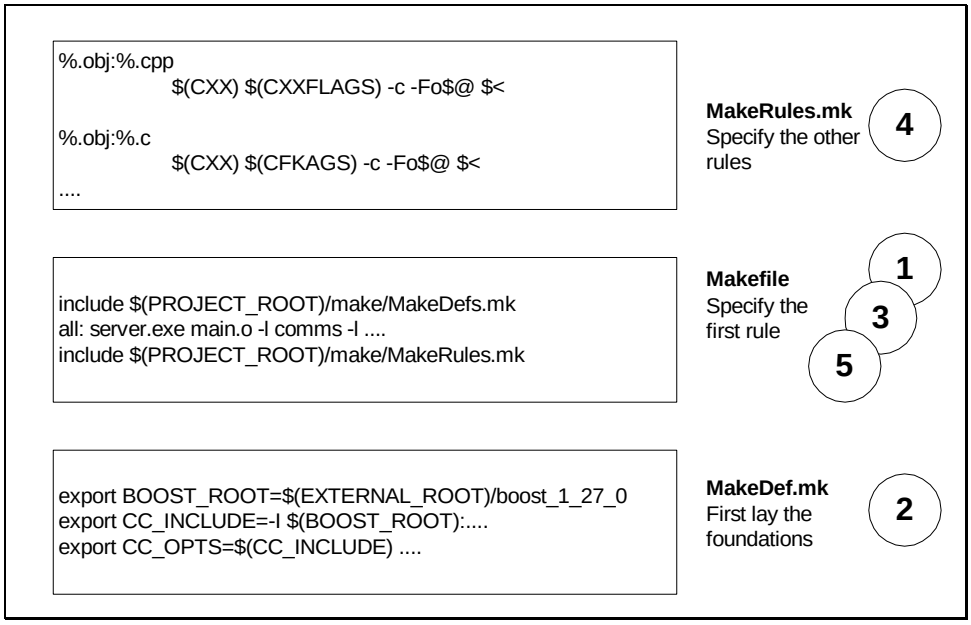

Figure 1 Makefile sandwich

From a command line, with our current directory containing the makefile shown in the above diagram, processing runs something like this:

1. Make is invoked and looks for `Makefile`, loading this file it starts to process

2. First include `MakeDefs.mk` this pulls in additional variable definitions. This file is not local so we must specify where to find it. If `PROJECT_ROOT` is not set it will fail. Importantly, it cannot contain any rules as we want `all` to be the first rule make encounters.
3. Encounter rule `all` and remember it is the first rule we have encountered.
4. Include `MakeRules.mk`, store rules "as is", only expand environment variables when the rules are encountered.
5. Execute the first rule, `all`, and any that are implied by it, when complete end.

Dependencies

One group of rules we need to specify is file dependencies, we need rules which say `Server.cpp` depends on `Server.hpp`, `Tcp.hpp`, `Utils.hpp` and so on. Given that a typical .cpp file may specify half a dozen or more .hpp file this can be a fairly large task so we don't want to write these rules ourselves. (If we don't specify these rules then the compiler won't get to recompile `Server.cpp` after a change to `Server.hpp`.)

Luckily help is at hand. GCC provides the several command line options (-M, -MG, -MM, etc.) to generate this information. Visual C++ provides the /FD switch for something similar - although I'll freely admit I've never actually used this in anger, I have looked at it well enough to say: it is not well documented and doesn't work like the GCC options.

My current make system uses makedepend, which originated at Tektronix and MIT as part of the X system. This comes as standard on some Unix versions, if you don't have it already, or are using Windows it is easy to track down a version on the net. Using the same program on Windows and Unix means there is no need for the makefiles to differ.

The default behaviour of makedepend is to append the generated rules to any makefile it finds in the current directory. This complicates matters as files will appear to change when there is no substantive change. A better way is to use makedepend to generate an additional makefile and include this. Since make will look at its own rules for any file it can't include this becomes almost trivial.

In the `MakeRule.mk` file we can add a new rule:

```
export DEP_FILE=depends.mk
$(DEP_FILE):
     @echo \# Generated dependencies file >$(DEP_FILE)
     @echo \# Never check this in to source control >>$(DEP_FILE)
     @echo \# Dependencies follow >>$(DEP_FILE)
     makedepend -f $(DEP_FILE) \
            -Y $(CC_OPTS) $(CC_INCLUDE) \
            $(SRCS) \
            -s"# Dependencies follow" > /dev/null 2>&1
     grep ".o:" $(DEP_FILE)
            | sed "s/\.o/\.obj/g" >>$(DEP_FILE)
```

Where SRC is a list of the source files to be processed, CC_OPTS specifies the compiler options and $(CC_INCLUDE) is a list of include paths.

In our local makefile we can add a new include after we include `MakeRules.mk`:

```
include $(DEP_FILE)
```

When make can't find `depends.mk` it will now know how to generate it from the rule in `MakeRules.mk`. Be warned: running makedepend can be time consuming.

The example rule has an additional grep command run once the dependencies have been generated. This is to cope with platform differences. On Unix object files are normally .o files, while on Windows they are normally .obj files. Being a Unix program makedepend will generate the rules in terms of .o files, for a Windows compile these rules are pointless, there will never be an .o file so the rules are never used. Unfortunately this means there are no rules for .obj files. Hence the grep and sed to create Windows equivalent rules. On a Unix system, the reverse is true and the .obj rules will be ignored.

Recursive and clean

Although not shown here it is advisable to include some house keeping targets. Typically a clean target will delete everything that has been generated, .obj's, .exe's, .lib's, etc. This can be drastic so some other targets like cleanlib and cleanobj can be included too which do lesser clean ups.

When we organise our directories as hierarchies we frequently end up with makefiles that don't run the compiler at all, instead they simply call several makefiles in sub-directories. There are two types of node in our hierarchy tree. Leaf nodes that contain source code and makefiles to compile the source, and intermediate nodes that serve to collect leaf nodes together.

Processing can be speeded up in these cases by skipping the inclusion `MakeRules.mk`.

Care needs to be taken to ensure that these intermediate directories pass through targets like clean. The clean rule in an intermediate makefile will look quite different to one in a leaf node, although both must share the same name.

To this end I've recently separated my clean rules into `MakeClean.mk` and `MakeCleanRecursive.mk`. `MakeClean.mk` can be included from `MakeRules.mk` so is available in all leaf nodes. Intermediate nodes don't include `MakeRules.mk` but instead include `MakeCleanRecursive.mk` to pass the target on to sub-directories.

Sometimes when recursing it is easier to use a bit of shell script, since make is happy to run a shell we can write:

```
clean: cleanlib

     for i in $(MODULES); do $(MAKE) -C $$i clean; done
```

The fact that we can recurse into our tree is thanks to our directory structure. If we had a simple flat structure with everything, applications and libraries hanging off a single root things would be more complicated.

Get source code

One of the task originally envisaged by makes' designers was getting source code from source code control before building, indeed, there are built-in rules for SCCS and RCS. Usually though it is better to treat the extraction of source code as one step, and the building as another even if this means having a shell script to run one command and then make.

This makes it easier for developers to work with make. When developing code you are unlikely to want make getting the latest version of files as soon as someone has checked them in. This is particularly true if using CVS where the very file you are working on may change if this happens.

Separating the get from the build means developers can choose when to update their tree. Simply running a build will not result in source code updates. Nor will we incur the time delay as the source control system checks for updates and retrieves any.

Although developers will usually update their tree at convenient internals to suit themselves batch builds should always build a fresh tree. It is a good test of a system to be able to completely delete a tree and rebuild it from scratch. Indeed, this is worth while exercise for developers to ensure they don't inadvertently forget to check-in some file.

Make Miscellany

? It is usually a good idea to disable to built-in rules. This can be done with the .SUFFIX directive – which itself has subtly changed its use over the years. Disabling the built-in rules assists with debugging (the make –d) and marginally improves performance.

? I've taken to including a help target (make help) in my systems. This normally takes the form of a help rule in a MakeHelp.mk file. Normally this will simply validate any external variables.

? The list of source files used for in building is normally referenced in several place. To save duplication it usually helps to lists these in one variable, e.g.

```
SRCS =  Configurator.cpp Interface.cpp main.cpp
```

? Frequently we want to create different build types for our compilations. Say a debug version and a release version. This is somewhat tricky as make expects to work in the current directory. My solution is to massage the filenames, this can be done with variable definitions given the list of source files:

```
# first change the suffix on the filenames
ifeq ($(COMPILER),msvc)
export OBJNAMES = $(SRCS:.cpp=.obj)
export OBJNAMES += $(CSRCS:.c=.obj)
endif
ifeq ($(COMPILER),g++)
export OBJNAMES = $(SRCS:.cpp=.o)
export OBJNAMES += $(CSRCS:.c=.o)
endif
# now append the build directory
export OBJS = $(addprefix $(BUILD_TYPE)/, $(OBJNAMES))
```

Where `BUILD_TYPE` would normally be set to Debug or Release in the environment.

? Remember to create directories before you try and put anything in them, this can be done with a rule for the directories themselves:

```
$(OUTPUT_DIR):
    mkdir -p $(OUTPUT_DIR)
$(BUILD_TYPE):
    mkdir -p $(BUILD_TYPE)
```

Generate what you can

Sometime we don't want to write in our chosen programming language. Some things are easier to represent in other forms, for example we use IDL to describe object interfaces, or we may want to encode the contents of a text file within source code, e.g. error messages which are defined externally, or localised message files.

In these cases it is useful to generate C++ code from another source and then compile it. Taking our error messages file this may contain an error number, sub-system code and a text message. Using a make rule we may specify a rule for converting text files to .cpp files, the rule could run a Python script. Once generated we would then compile the resulting .cpp file as normal.

The side box "Generating build numbers" provides an example of how we can use Python to generate C++ source code.

Just the beginning

Once you have your make system up and running there are a whole host of enhancements you may want to consider. Rather than go into details not here are some ideas:

? Always create a log file of message from the batch build – when it runs at night you need to know what happened.

? Make the build log public, e-mail to developers.

? Add automated tests to the build process – this is a good place to start automating your tests.

? Add a "package" target to the make system to collect all the files needed for an install. For Unix you may like to tar these up, for Windows you could automate your installer creation.

The example system I included with my article *Writing extendable software* (Overload 59) includes a simple make system along these lines and is available at http://www.allankelly.net. Time permitting I may upload another example.

Finally

Those who use Microsoft .dsp project files this may all seem excessively complex. However, .dsp files do not scale and do not work well on larger projects. Nor are they as rigorous and adaptable as makefiles.

Time spent developing a directory structure in depth, and a rigorous build system will result in a better defined project which can absorb additions and expansion more easily than one which is held together with sticky-tape and rubber bands. The rigors of this approach force problems to the surface.

By developing these aspects of the project fully we create strong logistics support for our development. This is all part of our strategy to create a software development process which can produce quality software, where every activity in the development process interlocks and supports the others.