



# **Building Large C++ Systems**

## **Make Tools**

Thomas Engelin

© Itancan Consulting AB 2008

# **1. Terminology**

## **1.1. Abbreviations**

gcc	GNU C Compiler
g++	GNU C++ Compiler
ld	Linker

## **1.2. Definitions**

Build system	Build script, make system, build tools
Build script	Script that calculates what to build and invokes the make system
Make system	A set of make files that invokes build tools
Build tools	Compiler and linker
ClearMake	Make- and audit tools from IBM Rational
GNU Make	GNU make tool

## **2. Abstract**

This document discusses building large C or C++ systems with GNU Make and ClearMake, bypassing command-line length limitations.

### **2.1. Who Should Read This?**

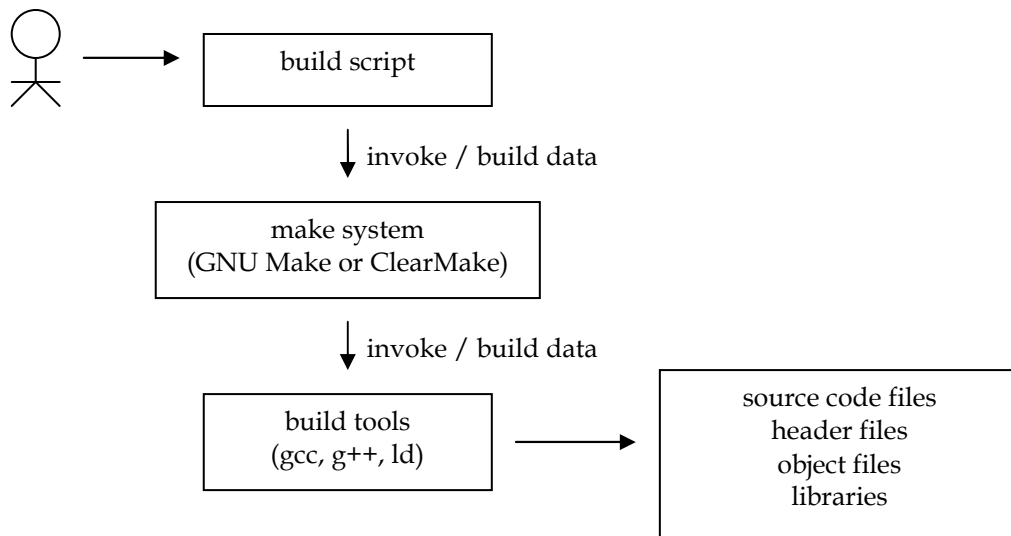
Build managers, software integrators, and software developers.

### 3. Problem Statement

Assume we have a very large C or C++ system with possibly thousands of source code files and header files that we build using either plain GNU Make or IBM Rational's ClearMake.

Also assume that the build system is highly dynamic and flexible, which means that the input and output of the build process varies a lot depending on what the user wants to build. This means that a lot of **build data** is passed between the components in the build system.

A common setup is to have a custom build script that calculates what to build and then invokes the make system, providing a range of input parameters:



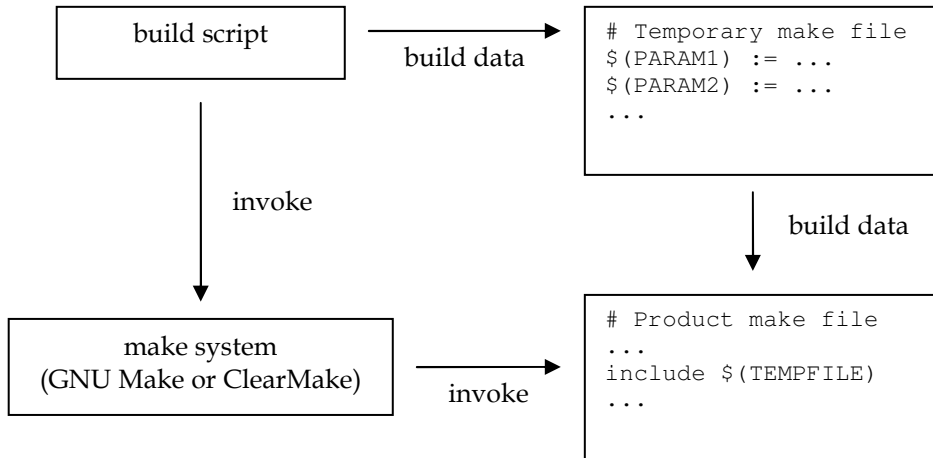
The core problem is that for large systems, the amount of data passed between the build script and the make system together with the data passed between the make system and the build tools, can be very large.

If the supporting operating system with its shells have a very limited command-line length, the build can start to fail when the system to build grows.

## 4. Passing Build Data

### 4.1. From build script to make system

The amount of data passed between the build script and the make system can easily be written to a temporary make file instead. The build script then just passes the path to this temporary make file to the make system, which includes it.



When the build is finished, the temporary make file can be deleted from the system, or kept for debugging or logging purposes.

## **4.2. From GNU Make system to build tools**

The amount of build data passed between the make system and the build tools can be quite big, especially when compiling with many include paths and when linking with many object- and library files.

The GNU compilers gcc and g++ provide the ability to specify input in a 'specs' file instead of on the command-line. Here is a standard way to invoke the compiler and linker:

```
g++ -o /home/user/source.o -c /home/user/source.cpp -I/home/user/
g++ -o ./myfile.o -c ./myfile.cpp -I/home/user/
g++ -o ./myprogram ./myfile.o /home/user/source.o
```

Using a 'specs' file, the compiler and linker can be invoked like this instead:

```
g++ -o /home/user/source.o -c /home/user/source.cpp -specs=./myspecs
g++ -o ./myfile.o -c myfile.cpp -specs=./myspecs
g++ -o ./myprogram ./myfile.o -specs=./myspecs
```

When compiling, the include paths can be placed in a temporary 'specs' file to reduce the length of the command line. When linking, object files and libraries can be put in the 'specs' file for the same reason. The 'specs' files can contain many things, such as build flags.

In this scaled-down example, the 'specs' file can look like this when compiling:

```
*cpp:
+ -I/home/user/
```

For the linking, the 'specs' file can look like this:

```
*endfile:
+ ./src/source.o
```

Multiple include paths and object files can of course be specified in this file:

```
*cpp:
+ -I/home/user/ \
+ -I/otherpath1/ \
+ -I/otherpath2/ \
+ ...
```

The syntax of this file is flexible, and it is possible to specify almost every aspect of compilation and linking.

The solution is to have the build script or the make system create temporary 'specs' files before the compiler and linker are invoked. When the build is finished, the temporary 'specs' files can be deleted from the system, or kept for debugging or logging purposes.

### **4.3. From ClearMake system to build tools**

Up until now, things have been quite straightforward and the only tricky thing has been to specify the correct syntax in 'specs' files read by gcc and g++.

When using ClearMake with support for wink-in, things get a bit more complex. The 'specs' files can easily be seen by ClearMake as part of the build recipe, which will affect the build avoidance algorithm.

To make wink-in work, the path to the 'specs' file must be equal between two builds. That is, we can no longer have 'specs' files use temporary file names, because then wink-in would never take place due to different text in configuration records.

The obvious solution is to store the 'specs' files in the current view at a fixed location. However, ClearMake is smarter than that. Even if the 'specs' file are stored in the view at the same location with the exact same \*nix file attributes, ClearMake understands that these actually are two different files, and wink-in will not take place.

So the 'specs' files have to be moved outside of the ClearCase VOB, possibly into a shared directory such as '/tmp'. With the 'specs' file outside of the VOB, ClearCase will see generated 'specs' files as one fixed file as long as they always use the same path.

But since the 'specs' files must share the same path, problems will occur when multiple builds are done against the same '/tmp' partition. With this path shared, a file locking protocol must be used, unfortunately with a possible degradation in build time. Also, the 'umask' when creating shared 'specs' files must be set up so that one user's build can delete the 'specs' file from another user's previous build.

Even with all these problems solved, there is another problem remaining. ClearMake can not compare the contents of two 'specs' files. Thus, differences between two 'specs' files cannot be detected by the build avoidance algorithm. If a target is built with two identical command-lines but is using different 'specs' files, ClearMake will make a faulty decision and perform a wink-in.

Using GNU 'specs' files in combination with ClearMake and wink-in seems impossible, or at least very tricky.

## **5. Conclusion**

The make system can easily be provided with build data from the build script using temporary make files.

The build tools can also be provided with GNU 'specs' files when using GNU Make or ClearMake without wink-in support. However, ClearMake without wink-in support shouldn't be a too common setup. If ClearMake together with wink-in is used, there seems to be no way to use 'specs' files at all.

## **6. References**

GNU 'specs' files, <http://www.gnuarm.org/pdf/gcc.pdf>