

Assignment 2 - Configuration tools

Thomas A. Grønneløv¹

Axel E. Jensen²

¹ di060169@diku.dk

² di060157@diku.dk

Abstract. When working with larger projects it becomes cumbersome to keep track of which files needs to be compiled and which files are dependent on other files. Different solutions exists and in this paper two utilities, **make** and **CMake**, are described and examples of usage are shown.

1. Introduction

Working with large software project often is a iterative process where programs are written (or rewritten), the source code is compiled and the programs are run for debugging or performance purpose. When some small part of the programs are changed maybe only a small part of the programs needs to be recompiled. It might also be necessary to compile with different options for optimization and debugging purpose and even for different platforms. This paper gives an overview of two tools, **make** and **CMake**, which handles some of the above tasks and discusses some of their advantages and disadvantages. The aim of the paper is mainly for people unfamiliar with the tool and should give basic understanding of the functionality and usefulness.

2. make

make is a high level tool which lets you build a *target* file from a number *prerequisite* (or dependent) files. The target file could be a executable file and the prerequisites could be C++ source files or the target file could be a PDF document and the prerequisites could then be a **T_EX** document.

When building a target **make** handles file dependencies and tracks which files have been modified since last build and therefore tracks which files needs to be recompiled. For each target **make** invokes a series of commands, typically invoking a compiler with given options and files which needs compiling.

Targets, prerequisite and which commands applies are store in files called *Makefiles*. Store the Makefile in the same directory as the source files. Below is shown an example of a simple Makefile.

```
...
hello: hello.o
```

```

        cc -o hello hello.o

hello.o: hello.c
        cc -c hello.c
...

```

In the first two lines **here** is the target, **hello.o** is the prerequisite and **gcc -o hello hello.o** is the command invoked. The order of the two targets does not matter to **make**. There should be a tab in front of the command and no where else.

2.1 Variables

Using variables simplifies the management and updating of Makefiles. An example of using variables follows:

```

...
OBJECTS = main.o matrix.o vector.o

transform: $(OBJECTS)
        cc -o transform $(OBJECTS)
...

```

In this way it is simple to add another object file to the project. It needs only to be added to the variable **OBJECT**. Another useful use of variables is the ability to override them when running **make**.

In a Makefile:

```

...
CFLAGS = -g

matrix.o: matrix.c
        cc -c $(CFLAGS) matrix.c
...

```

Overriding **CFLAGS** can be done in the following way:

```
make CFLAGS='-O'
```

2.2 Pattern rules

Another useful feature in **make** is the use of pattern rules. This allows you to specify targets or prerequisites which matches certain patterns and you do not need to specify every single file.

```

...
%.o : %.c
$cc -c $(CFLAGS) $< -o $@
...

```

This rule compiles every **.c** file into a corresponding **.o** file. This also shows two automatic variables **\$<** and **\$@** which represents name of the target and of the prerequisite.

For more in depth documentation on GNU Make please refer to the Gnu Make Manual [4].

2.3 Usage of *make*

Even though **make** is able to solve many of the problems related to handling large project it has some drawbacks. One item of complaint is the syntax of **make**, which is categorized as being "really stupid" [5] or maybe somewhat more objective observed as limited, "...obscure and, consequently, more difficult than you might think" [2].

Even though **make** is capable of building and updating libraries and even building targets in parallel, it lacks (to our knowledge) of a scheduling mechanism required when building time is of considerable magnitude.

Although **make** exist for different platforms like Linux, BSD and Windows, **make**-based builds are not necessary portable. As mentioned in [2] this is due to "The problem is that the Make tool ... has to rely on shell commands and on features of the filesystem. These two are notorious sources of platform incompatibilities."

Software projects might also be developed in parallel by different people working in different ways. Some developer are using Integrated Developments Environment (IDE) and the IDE might administrate source file in different ways. In the following a tool, **CMake**, that might overcome this problem is described.

3. CMake

CMake is a system, that manages the build process across different platforms. It is able to produce Makefiles on Unix platforms and /projects on MS Visual C++. **CMake** supports compilation of source code (using native build tools like **make** and **MVSC**), creation of libraries, instantiation of templates among many other things.

A. Neundorf mentions in [3] **CMake** as an alternative to autotools and similar build systems. And this alternative is cross platform different from most other.

CMake uses simple configuration files called *CMakeLists.txt* files, which are placed in each (sub)directories containing source files. The *CMakeLists.txt* contains a number of *commands*. The following example shows a simple *CMakeLists.txt* and how a Makefile is generated on a Linux system. *CMakeLists.txt* contains the following two commands:

```
PROJECT (MyProject)
ADD_EXECUTABLE(MyProgram hello.c)
```

and **CMake** is invoked, a Makefile is generated and subsequently used by **make**:

```
SHELL>cmake .
SHELL>make
```

In the following sections, some common use commands are explained.

3.1 Listfile Commands

```
PROJECT (MyProject)
ADD_EXECUTABLE(MyProgram main.c vektor.c matrix.c)
```

PROJECT() defines the projects name in MVSC and ADD_EXECUTABLE() adds an executable to the list of target. Here, the program requires a few files, which also could have been place in a SET:

```
PROJECT(MYPRO)
SET(MYPRO_SRC main.c vector.c matrix.c)
ADD_EXECUTABLE(MyPro $(MYPRO_SRC))
```

Analogous to variables in `make`. Sets can be manipulated e.g. depending on the platform:

```
...
IF(WIN32)
SET(MYPRO_SRC $(MYPRO_SRC) WinSupport.c)
ELSE(WIN32)
SET(MYPRO_SRC $(MYPRO_SRC) UnixSupport.c)
ENDIF(WIN32)
```

A few other usefull commands :

```
SUBDIRS()
ADD_LIBRARY()
```

CMake works recursively and the subdirectories are listed in SUBDIRS(). In the subdirs an additional CMakeLists.txt is placed. ADD_LIBRARY() adds a library to the project and specifies the source files.

Please refer to documentation in [1] for more in depth documentation (or try `cmake --help-html [file]`).

3.2 Usage of CMake

As CMake uses native build tools e.g. `make` one would still have some of the same problems related to the native build tool. Some problems are overcome though. Maintaining Makefiles and handling the somewhat difficult syntax of Makefiles are handled by CMake. Cross platform support is also handled by CMake. But compiling in parallel would benefit from scheduling mechanism not supported by CMake.

References

- [1] I. Kitware, <http://www.cmake.org/HTML/Documentation.html>, WWW (2006).
- [2] A. Neagu, <http://freshmeat.net/articles/view/1702/>, WWW (2005).
- [3] A. Neundorf, <http://lwn.net/Articles/188693/>, WWW (2006).
- [4] P. Smith, <http://www.gnu.org/software/make/manual/make.html>, WWW (2006).
- [5] M. Welsh, M. K. Dalheimer, and L. Kaufman, *Running Linux*, O'Reilly (1999).