Søren Hansen <sha@ase.au.dk>

Building C/C++ programs for host n target

V2.2

Introduction

In this exercise you will use different ways of compiling a program for Linux. The programs you create are not the issue in this exercise – it is the process of compiling them that is in focus. It is important that you finish this lab exercise as it is the basis for next week's exercise. Furthermore all subsequent exercises require that you use makefiles.

Prerequisites

In order to complete this exercise, you must:

- Have completed the *Getting set up* exercise, thus a working Linux (in a VMWare)
- Have access to a *target* - DevKit8000.

Exercise 1 The Hello World program

In the file `hello.cpp` write a small `"Hello World!"` C++ program. Use direct compiler invocation of the compiler `g++` to compile your program to an executable `hello`. Correct any errors you may have, then execute your program.

Exercise 2 Makefiles - compiling for host

Exercise 2.1 Makefiles???

Writing *makefiles* is in itself writing in a scripting language meaning that one has to abide by the language rules that apply.

Therefore before the first makefile is to be written certain keywords and concept must be known.

Answer the following questions and remember to use code snippets if it serves to pave the way for better understanding.

- What is a target?
- What is a dependant and how is it related to a target?
- Does it matter whether one uses *tabs* or *spaces* in a `makefile`?
- How do you *define* and *use* a variable in a `makefile`?
- Why use variables in a `makefile`?
- How do you use a created `makefile`?
- In the `makefile` scripting language they often refer to *built-in* variables such as these
  - `$@`, `$<` and `$^` - explain what each of these represent.
  - `$(CC)` and `$(CXX)`:
    * What do they refer to?
    * How do they differ from each other?
  - `$(CFLAGS)` and `$(CXXFLAGS)`:

    * What do they refer to?
    * How do they differ from each other?

- What does $\$(SOURCES : .cpp = .o)$ mean? Any spaces in this text???

Exercise 2.2 Using makefiles - starting out

Write a `makefile` for the program `hello` you created in the first exercise. Add a target *all* that compiles your program, furthermore use variables to specify the following:

- The name of the executable
- The used compiler.

Compile your program using `make` and execute it.

Add two targets to your makefile; *clean* that removes them all object files as well as the executable. Add a target *help* that prints a list of available targets[1]. Remember to verify via tests that all three targets do as expected.

Exercise 2.3 Program based on multiple files

Exercise 2.3.1 Being explicit

Create a simple program `parts` consisting of 5 files:

- `part1.cpp`
  contains 1 simple function `part1()` that prints `"This is part 1!"` on `stdout`
- `part1.h`
  contains the definition of `part1()`
- `part2.cpp`
  contains 1 simple function `part2()` that prints `"This is part 2!"` on `stdout`
- `part2.h`
  contains the definition of `part2()`
- `main.cpp`
  contains `main()` which calls `part1()` and `part2()`

Create a makefile for `parts`. As in Exercise 2.2 and specify the executable and the used compiler by means of variables. Add targets *all*, *clean* and *help*[2] as in Exercise 2.2.

Exercise 2.3.2 Using pattern matching rules

The `makefile` created in the previous exercise is very explicit and rather large. In this exercise the idea is to use the same but shrink it down and make it less error prone. In `make` the solution is to use *pattern-matching* which has special syntax involving the % character for representing wildcards[3]. In other words, one writes a general *rule* that applies to many situations alleviating the need to write a rule for each and every file.

In this version of the `makefile` two extra variables are needed:

[1] Do note that `make` does *not* have in-built facility to do this.
[2] Prints out a help guide as to how the makefile in question can be used. Yep, `make` does **not** have some built-in cool thingy,. You have to write it yourselve... This is here the command `echo` comes in handy
[3] If unsure read *again* the text regarding *pattern-matching* in `make` - one could also JFGI...

- Source files
- Object files (acquired from the source file variable - how?)

Write an pattern matching rule that creates *object* files based on our *cpp* files.

What makes this an improved solution as opposed the previous one?

Exercise 2.4 Problem...

The below `makefile` snippet compiles and produces a working executable. This is obviously assuming that the said files exist and are adequately sane. In this particular scenario it is assumed that the following files exist:

- server.hpp & server.cpp
- data.hpp & data.cpp
- connection.hpp & connection.cpp

Listing 2.1: Simple makefile creating a simple program executable called prog

```
EXE=prog
OBJECTS=server.o data.o connection.o

$(EXE): $(OBJECTS)
     $(CXX) -o $@ $^
```

Questions to consider:

- How are the source files compiled to object files, what happens[4]?
- When would you expect `make` to recompile our executable `prog` - be specific?
- Make fails using this particular `makefile` in that not all dependencies are handled by the chosen approach. Which ones are not[5]?
- Why is this dependency issue a serious problem?

Exercise 2.5 Solution

Analyze the listing 2.2.

Listing 2.2: Using finesse to ensure that dependencies are always met

```
SOURCES=main.cpp part1.cpp part2.cpp
OBJECTS=$(SOURCES:.cpp=.o)
DEPS=$(SOURCES:.cpp=.d)
EXE=prog
CXXFLAGS=-I.

$(EXE): $(DEPS) $(OBJECTS)    # << Check the $(DEPS) new dependency
  $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)
```

[4]Use simple deduction first and then seek an answer by reading chapter 10.2 in the pdf file `http://www.gnu.org/software/make/manual/make.pdf`

[5]If it isn't obvious what the problem is, then try changing the various files and see what happens

```

# Rule that describes how a .d (dependency) file is created from a .cpp
    file
# Similar to the assigment that you just completed %.cpp -> %.o
%.d: %.cpp
  $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $^ > $@

-include $(DEPS)
```

Describe and verify what it does and how it alleviates our prior dependency problems!

In particular what does the command `$(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $(SOURCES)` do? See your `man` files... Hint try to run the command part thereof by hand and examine its output.

Exercise 3 Cross compilation & Makefiles

Cross compiling is simple in the sense that `gcc`, in our case, is exchanged with another `gcc` that supports the desired target, e.g. `arm-devkit-gcc`. At this point we want to develop the final `makefile` that handles all the issues encountered in one go.

Take your starting point in the listing 3.1 and finalize the missing parts such that we get a `makefile` that attains the desired functionality.

Listing 3.1: Handling cross compiling properly

```
SOURCES=main.cpp part1.cpp part2.cpp
OBJECTS=$(SOURCES:.cpp=.o)
DEPS=$(SOURCES:.cpp=.d)
EXE=prog
CXXFLAGS=-I.

# Making for host
# > make ARCH=host
ifeq (${ARCH},host)
CXX=g++
BUILD_DIR=build/host
endif

# Making for target
# > make ARCH=target
ifeq (${ARCH},target)
CXX=arm-devkit-g++
BUILD_DIR=build/target
endif


$(EXE): $(DEPS) $(OBJECTS)    # << Check the $(DEPS) new dependency
  $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)

# Rule that describes how a .d (dependency) file is created from a .cpp
    file
```

```
# Similar to the assigment that you just completed %.cpp -> %.o
${BUILD_DIR}/%.d: %.cpp
	$(CXX) -MT${BUILD_DIR}/$(@:.d=.o) -MM $(CXXFLAGS) $^ > $@

ifneq ($(MAKECMDGOALS),clean)
-include $(DEPS)
endif
```

Things to alter:

- *Objects* placement - *now*
  As it is now, where are the objects placed? Why is this bad?
  Hint: See listing 2.2 where *pattern-matching* has been used.
- *Objects* placement - *after change*
  Where are they to be placed now?
  Explain how this is achieved.
- *Program file*
  Is the current placement the correct one? Hardly; what to do and where to place it?

Hints: Checkout the `make` command `$(addprefix)`. Ensuring that the dependency files are generated in the correct spot has already been fixed. Furthermore what should the target be???
Suggestion:

- Place all generated object files in `build/target` or `build/host` (this is what has already begun in the above `makefile` listing) respectively.
- Place the executable in `bin/target` or `bin/host` respectively.

Exercise 4 Libraries (optional)

Exercise 4.1 Using libraries

In numerous situations the functionality you need to use is placed in a library for everyone to use.

A lot of commandline programs are pretty boring from a TUI[6] point of view. Just simple read lines and printfs nothing fancy. Sometimes, though, one would like it to be more fancy. This is where *ncurses* may help out[7].

Find the *hello world* program or something similar on their web page `http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/` and remember to link with their library (how, is also shown on their web page). Their *hello world* program is nothing fancy, but it illustrates some simple features and importantly forces you to link a library to your program.

Pick out one of your already created `makefiles` and modify it such that you may link and afterwards run the program.

- How do you link a library to a program?

[6]Text User Interface
[7]Actually this might be beneficial for your semester project for testing purposes.

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

- When linking a library to a given program one obviously need to know the name of the file. However, is the name of the file as found on the disk exactly the same characterwise as when supplying it to gcc?

Do note that the library may not be installed. In that case, you need to install the ubuntu package *libncurses5-dev*.

Exercise 4.2 Creating your own static library

Make `part2` from exercise 2.3 a *static library*. Make sure that your program links with the library.

This exercise might be very interesting to complete in relation to your project.

Questions to consider:

- How do you make a *static library*?
- Why would you do it?
- Which changes are needed in our `makefile` to facilitate this?
- In this exercise the *static library* is an integral of the same `makefile`. How do you think a more realistic solution would like? And which changes to the `makefiles` would this encompass?