Søren Hansen <sha@ase.au.dk>

Building C/C++ programs for host n target

V2.3

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* - *Raspberry Pi Zero W*.

Exercise 1 The first Makefile

Exercise 1.1 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 1.2 Makefile basics

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 1.3 Writing the makefile

Write a `makefile` for the program `hello` you just created. In the presentation you will find a makefile snippet you can rip-off.

Test and verify that make does what you it to do :-)

Exercise 2 Makefiles - compiling for host

Exercise 2.1 Using makefiles - Next steps

Rewrite your first `makefile`. 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 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 2.2 Program based on multiple files

Exercise 2.2.1 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 2.1 and specify the executable and the used compiler by means of variables. Add targets *all*, *clean* and *help*[2] as in Exercise 2.1.

[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

Exercise 2.2.2 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.
To achieve this *pattern-matching* will be employed. It has a 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:

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

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

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

Exercise 2.3 Problem...

The below `makefile` compiles and produces a working executable (*IT DOES*). 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 (You may need to create some to figure out what happens...):

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

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

```
1 EXE=prog
2 OBJECTS=server.o data.o connection.o
3
4 $(EXE): $(OBJECTS)
5      $(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 / precise with respect to file names?
- Make fails using this particular `makefile` in that not all dependencies are considered by the chosen approach. Which ones are not[5]?
- Why is this dependency issue a serious problem?

[3] If unsure read *again* the text regarding *pattern-matching* in `make` - one could also JFGI...
[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

Exercise 2.4 Solution

Analyze the listing 2.2.

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

```
1  SOURCES=main.cpp part1.cpp part2.cpp
2  OBJECTS=$(SOURCES:.cpp=.o)
3  DEPS=$(SOURCES:.cpp=.d)
4  EXE=prog
5  CXXFLAGS=-I.
6
7  $(EXE): $(DEPS) $(OBJECTS)   # << Check the $(DEPS) new dependency
8    $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)
9
10 # Rule that describes how a .d (dependency) file is created from a .cpp
       file
11 # Similar to the assigment that you just completed %.cpp -> %.o
12 %.d: %.cpp
13   $(CXX) -MT$@ -MM $(CXXFLAGS) $< > $@
14   $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $< >> $@
15
16
17 -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) $(SOURCES)` do? See your `man` files... Hint try to run the command part by hand and examine its output. Note that the option `-MT$(@:.d=.o)` does not do much here, however it is important onward which is why you need to figure out how it works.

Exercise 3 Cross compilation & Makefiles

Its important that you have completed the "Connecting to target" before commencing onward from here!

Exercise 3.1 First try - KISS

Cross compiling is simple in the sense that `g++`, in our case, is exchanged with another `g++` that supports the desired target architecture, e.g. *ARM* and thus the compiler `arm-rpizw-g++`. That being said lets revisit exercise 2.1 in which a simple `makefile` was made.

Start by making a copy of this `makefile` called `makefile.arm` and change it such that it uses the arm compiler. Having done that invoke it using `make`.

Consider:

- Do you have to do something special to invoke this particular `makefile`?
- At this point we have two makefiles in the same dir. How does this present a problem in the current setup and how are you forced to handle it(Hint: *Think object / bin files*)?

Exercise 3.2 The full Monty - Bye bye KISS

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

```
1  SOURCES=main.cpp part1.cpp part2.cpp
2  OBJECTS=$(SOURCES:.cpp=.o)
3  DEPS=$(SOURCES:.cpp=.d)
4  EXE=prog
5  CXXFLAGS=-I.
6
7  ARCH?=x86-64
8
9  # Making for x86-64 e.g. x86-64 (the architecture employed)
10 # > make ARCH=x86-64
11 ifeq (${ARCH},x86-64)
12 CXX=g++
13 BUILD_DIR=build/x86-64
14 endif
15
16 # Making for architecture
17 # > make ARCH=arm
18 ifeq (${ARCH},arm)
19 CXX=arm-rpizw-g++
20 BUILD_DIR=build/arm
21 endif
22
23
24 $(EXE): $(DEPS) $(OBJECTS)   # << Check the $(DEPS) new dependency
25   $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)
26
27 # %.cpp -> %.o needs to be added! Target is NOT just %.o...
28
29 # Rule that describes how a .d (dependency) file is created from a .cpp
     file
30 # Similar to the assigment that you just completed %.cpp -> %.o
31 $(BUILD_DIR)/%.d: %.cpp
32   $(CXX) -MT$@ -MM $(CXXFLAGS) $< > $@
33   $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $< >> $@
34
35 ifneq ($(MAKECMDGOALS),clean)
36 -include $(DEPS)
37 endif
```

Things to alter:

- *Objects* placement - *now*
  As it is now, where are the objects placed (main.o etc)? Why is this bad?
  Hint: See listing 2.2 where *pattern-matching* has been used.

- *Objects* placement - *after desired change* e.g. *new placement*
  Where are they to be placed now (See .d file generation placement in 3.1)?
  Explain how this is to be achieved (Hint: *Think target name*).
- *Program file*
  Is the current placement the correct one? Hardly; what to do and where to place it?
- Note the extra command for building .d files. Answering this one is a bonus question. :-)
  But use it never the less.

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/arm` or `build/x86-64` (this is what has already begun in the above `makefile` listing) respectively.
- Place the executable in `bin/arm` or `bin/x86-64` respectively.

Exercise 4 Improving code quality...

Different tools exist to aid you in writing better code quality. In this exercise we will use two different ones and employ a simple approach at that. You *will* gain significantly by getting to know these and employ them in the course as well as in your project.

The exercises in this section is mostly about trying this out and having these tools at your disposal for the rest of this semester...

Exercise 4.1 clang-format

`clang-format` simply formats your code, such it conforms to some predetermined layout. IT does not change casing or any other aspect of your code that would imply that your code has actually changed. E.g. needs recompilation.

Per default it searches for a file named `.clang-format` up through the file hierarchy. As it is, one has been placed in ~.

It's contents is:

Listing 4.1: clang-format config

```
1  AlignTrailingComments: true
2  AllowAllParametersOfDeclarationOnNextLine: true
3  AllowShortFunctionsOnASingleLine: true
4  AllowShortIfStatementsOnASingleLine: true
5  AllowShortLoopsOnASingleLine: true
6  BreakBeforeBraces: Allman
7  ConstructorInitializerAllOnOneLineOrOnePerLine: true
8  IndentWidth: 2
9  AlignConsecutiveDeclarations: true
10 AlignConsecutiveAssignments: true
```

Add the following to your `Makefile` to simplify usage:

Listing 4.2: Using clang-format in a Makefile

```
format: $(SOURCES:.cpp=.format)
%.format: %.cpp
  @echo "Formatting file '$<'"...
  @clang-format -i $<
  @echo "" > $@
```

Your task: Try to run it on your code and explain what you experience. Just that simple, nothing more!

Exercise 4.2 clang-tidy

`clang-tidy` checks whether your code follows certain standards as well as whether you employ some bad practices. IT does not change your code (how we use it) but rather output errors when your code does not conform or uses bad practises.

Per default it searches for a file named `.clang-tidy` up through the file hierarchy. As it is, one has been placed in ~.

It's contents is:

Listing 4.3: clang-tidy config

```
Checks: '-*,readability-identifier-naming'
CheckOptions:
  - { key: readability-identifier-naming.NamespaceCase,        value:
      CamelCase  }
  - { key: readability-identifier-naming.ClassCase,            value:
      CamelCase  }
  - { key: readability-identifier-naming.PrivateMemberSuffix, value: _
            }
  - { key: readability-identifier-naming.StructCase,           value:
      CamelCase  }
  - { key: readability-identifier-naming.FunctionCase,         value:
      camelBack  }
  - { key: readability-identifier-naming.VariableCase,         value:
      camelBack  }
  - { key: readability-identifier-naming.GlobalConstantCase,   value:
      UPPER_CASE }
```

Add the following to your `Makefile` to simplify usage:

Listing 4.4: Using clang-tidy in a Makefile

```
tidy: $(SOURCES:.cpp=.tidy)
%.tidy: %.cpp
  @echo "Tidying file '$<'"...
  @clang-tidy  $< -- $(CXXFLAGS)
  @echo "" > $@
```

Your task: Try to run it on your code and explain what you experience. Just that simple, nothing more!

Exercise 4.3 Makefile QoL

To ensure that neither `clean` nor `format` or `tidy` invoke dependency generation remove the following from your `Makefile`:

Listing 4.5: Old approach

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

AND add this instead:

Listing 4.6: New approach ensuring that dependency generation is not performed when targets are one of *clean*, *format* and *tidy*

```
1 ifneq ($(filter-out clean format tidy,$(MAKECMDGOALS)),)
2 -include $(DEPS)
3 endif
```

Exercise 5 Libraries

Exercise 5.1 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?
- When linking a library to a given program one obviously needs to know the name of the file. However, one thing is the actual file name another is how it is denoted when linking... Whats the difference?

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

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

Exercise 5.2 Creating your own static library (OPTIONAL)

Make `part2` from exercise 2.2 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 part of the same `makefile`. How do you think a more realistic solution would like? And which changes to the `makefiles` would this encompass?