Building C++ programs for host

shan@iha.dk
V1.4

Introduction

In this exercise you will use different ways of building a program for Linux. The programs you create are not the issue in this exercise – it is the process of building 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)

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 build your program to an executable `hello`. Correct any errors you may have, then execute your program.

Exercise 2 Using makefiles

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.

Build 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. Remember to verify via tests that all three targets do as expected.

Exercise 3 Program based on multiple files

Exercise 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 and specify the executable and the used compiler by means of variables. Add targets *all*, *clean* and *help*[1] as in Exercise 2.

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

- 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 4 The missing piece

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 4.1: 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[2]?
- In which circumstances should `make` rebuild our executable `prog`?
- Make fails using this particular file in that not all dependencies are handled by this approach. Which ones are not[3]?
- Why is this dependency issue a serious problem?

[1] Prints out a help guide as to how the makefile in question can be used. This is here the command `echo` comes in handy

[2] 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`

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

Analyze the listing 4.2.

Listing 4.2: Using finesse to ensure that dependencies are always met

```
1  SOURCES=file1.cpp file2.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   $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $^ > $@
14
15 -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 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[4] 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.

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?
- Is the name of library file wise the when you link with it on using gcc?

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

[4] Text User Interface

Exercise 5.2 Creating your own static library - OPTIONAL

Make `part2` from exercise 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?