TDT4200 - Assignment 0
Compiling and run your C code

Raymond Toft

2018

1 Introduction

This tutorial is ment as an guidance for those of you that may have limited experience with working with source files in c and c++. From previous experience, many students also have limited knowledge of Linux operating systems. We will here go through the basic commands for navigating with the command line and how to compile and run your code. In this course we will be using a tool named CMake to compile our code for the assignments. We will only briefly show you how this works and how you may apply it. It is advised to search for additional information on how compiling and CMake works to develop a deeper insight.

2 The command line

For the Unix operating systems we have the command line in the terminal which we will use to navigate and execute our programs. To navigate in the directory hierarchy we use the command *cd directoryName*, where *directoryName* is the designated directory we want to enter. By entering *cd ..* or *cd* we can navigate back a level or to the root level, respectively. We can also navigate through multiple layers simultaneously be using / in between each directory name. To view the contents in a directory we use the command *ls*.

We can make a new directory with the command *mkdir myNewDirectory*. In addition we can remove files with *rm myFileName*. If we try to remove a directory we get a warning *rm: cannot remove 'myDirectory/: Is a directory*. To be able to remove a directory form the command line, we must add a flag to force the deletion: *rm -r myDirectory*.

The terminal and command line, along with the mentioned commands is illustrated in Figure 1.

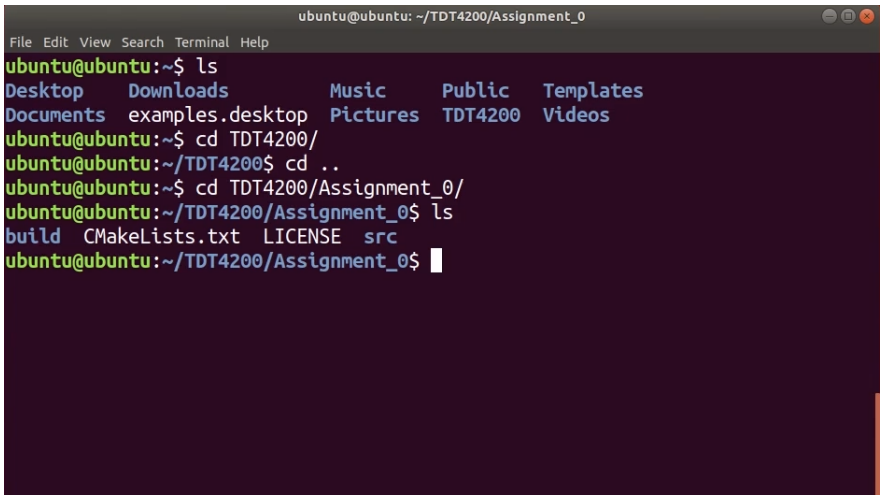

Figure 1: The command line in the terminal. The command *ls* displays the content of the current directory. Use *cd* to navigate between the directories.

3 Compiling and building source code

To be able to run and execute our C and C++ programs, we need to first compile and build them first. We can compile the source files manually with GCC, by using the following command

```
$ gcc -Wall main.c myFunctions.c -o myProject
```

This compiles the list of source code, *main.c* and *myFunctions.c*, to machine code and stores it in an executable file *myProject*. This output file is specified by using the *-o* command. The name of the -o file does not need to be the same as the source file. If the o-file is omitted, the output is written to a default file called *a.out*. This option can be used for debugging purposes. We will not go further into this, but encourage you to investigate it further. As a note on the compiling of source files, we do not specify header files in the list of files to be compiled. They are already included in the source files by the directive *#include "myHeaderFile.h"*.

In the space between $ *gcc* and the source file, we can specify compiler flags. In the example above we have used *-Wall*, witch turns on all the most commonly used compiler warnings. This should always be present while compiling, since GCC does not produce any warnings unless they are enabled. These compiler warnings are essential in detecting problems when programming in C and C++.

Now that we have compiled the program, ve type the path name of the executable to run the program:

```
$ ./myProject
```

This loads the executable file into memory and causes the CPU to begin exe-

cuting the instructions. The path ./ refers to the current directory.

While this is a simple process, it easily becomes tedious when the project grow with many source files and a variety of compiler options. Te ease this process, we use scripts to organize the code compilation. These scripts are called Makefiles.

3.1 Make and Makefiles

Makefiles are a simple way to organize code compilation. These scripts are executed with a tool called *make*, which is an automation tool for compiling and building applications. The compilation command above can be written as a Makefile:

```
CC = gcc
CFLAGS = -Wall
myProject: main.c myFunctions.c
    $(CC) $(CFLAGS) -o myproject main.c myFunctions.c -I.
clear:
    rm -f myProject
```

Here, we see that the compiling process easily can be organized into several "rules". In this example we have specified two "rules", *myProject* and *clear*. These are commands that we can use along with make to compile the code or clear previous built code.

To compile and build our program we type either of the following commands:

```
$ make

$ make myProject
```

Either of the above commands will compile your code. The difference is when you have several "rules" in you Makefile. By only typing *make*, all the compilation rules are executed. On the other hand, if you type *make myProject*, only that rule will be compiled. To delete the compiled execution files, we can run the final rule *clear*. It is common practice to include a rule that delete o-files and other output files to ensure a clean directory when a fresh compilation and execution is needed.

Makefiles and make are a powerful tool that have many options and possibilities. We encourage you to read more about Makefiles to get a better understanding of how they work and what options they offer.

If your system by any chance should not have make installed, run the following command to install make, gcc and other useful tools:

```
$ sudo apt-get install build-essential
```

Now that we have covered how to compile and build your code both manually and by Makefiles, we will now introduce how we can take a step further and automate the setup of the Makefiles.

4 CMake

For the assignments we will be using CMake to built our source code. CMake is an extensible, open-source system that manages the build process in an operating system, and in an compiler-independent manner. To install CMake, type the following in the command line:

```
$ sudo apt-get install cmake
$ cmake --version
```

When CMake is executed, it will look for a textfile named *CMakeLists.txt*. This file contains all the instructons for CMake. The basic CMakeLists.text file may look like this:

```
cmake_minimum_required(VERSION 2.8.9)
project(myProject)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}" -Wall)
add_executable(myProject main.c myFunctions.c)
```

The first line sets the minimum version of CMake for this project. By providing a version number we allow for future support for our build environment. By default you should use the current version of CMake on your system, as we determined after the installation above. The second line sets the project name for our build. On the third line we specify the compiler flags. Finally we request with the *add_executable()* that an executable to be built using the two source files. The first argument is the name of the executable to be built, namely the previous specified project name.

Now that we have made a CMakeList.txt file, we want to bould our system. It is standard practice to first create a build directory and call CMake from there:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

The third command launches CMake from the directory, and looks for the CMakeList.txt file in the above directory. The generated Makefile and other build files is now in the current build directory. With the final command we compile the source files with make. In this example we can now execute the runfile by typing **./myProject**.

Unfortunately we have not a "clean" command in CMake as we did with the Makefile and make. To remove all related build files, we must remove the *build* directory with the aforementioned command: *rm -r build*.

4.1 CMake in the assignments

We have have now briefly demonstrated how we compile and build source code and execute the runfile. In the assignments of this course you will receive a

hand out *CMakeLists.txt* file. The source coude will be in its own directory, and is added in *CMakeLists.txt*. In addition there is a build directory wher you will call the *CMakeLists.txt* from, and build the project as illustrated above. It is worth noting that the executable file will be placed in a subdirectory *cintro* of the *build* directory. You can either navigate into this directory and run the executable file with *./cintro*, or you can run the file from the build directory with

```
cintro/cintro
```