Lecture 17: Automated Compilation

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/17-automated-compilation.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we introduce a tool called `make`, that allows to automate the compilation of programs composed of multiple and potentially many source files.

Incremental Build

When we have a program composed of several modules, there is no need to rebuild the entire program when just one or a few source files are changed. Let's take the example from the following lecture in which we build our program from 3 source files, `network.c`, `parser.c` and `main.c`. A possible scenario is:

```
# Initial build:
$ gcc -c network.c -o network.o
$ gcc -c parser.c -o parser.o
$ gcc -c main.c -o main.o
$ gcc main.o network.o parser.o -o prog

# Assume we update parser.c, to rebuild we don't need to recompile everything:
$ gcc -c parser.c -o parser.o
$ gcc main.o network.o parser.o -o prog

# Now we update parser.h, remember it's included in both parser.c and main.c, so:
$ gcc -c parser.c -o parser.o
$ gcc -c main.c -o main.o
$ gcc main.o network.o parser.o -o prog
```

We have an initial build in which we compile each file and link the object files into the program. Now assume we update `parser.c`, to rebuild the entire program we only need to recompile that source file and relink all object files. We don't need to recompile `network.c` and `main.c`. Same thing if we modified `parser.h`, we only need to recompile the files including that header, `parser.c` and `main.c`, then relink. Keeping track of all these dependencies manually is difficult though. Thankfully there exists a tool that can automatically manage all that process.

Makefiles: Automated Build and Dependency Management

This tool is called `make`. It relies on configuration files named Makefiles. They live alongside the sources, describing the build and its dependencies, following a particular format.

Let's consider the build dependencies for our server application example:

- The final executable `prog` is produced by the link stage that requires the 3 object files, `network.o`, `parser.o`, and `main.o`.
- `network.o` is the result of compiling `network.c`, which also include `network.h`.
- `parser.o` is the result of compiling `parser.c` that also includes `network.h` and `parser.h`.
- Finally, `main.o` is the result of compiling `main.c`, which includes both headers.

Now that we have reasoned about the dependencies we know what needs to be done when a given file is modified. For example, if `parser.h` is modified, then `main.o` and `parser.o` need to be rebuilt, and `prog` needs to be relinked.

An Example of Makefile

We describe these dependencies in a file named `Makefile`:

```
# The first rule is executed when the
# command make is typed in the local folder:
all: prog

# executable deps and command to build it:
prog: main.o network.o parser.o
    gcc main.o network.o parser.o -o prog

# network.o deps and command to build it:
network.o: network.c network.h
    gcc -c network.c -o network.o

parser.o: parser.c parser.h network.h
    gcc -c parser.c -o parser.o

main.o: main.c network.h parser.h
    gcc -c main.c -o main.o

# Special rule to remove all build files
clean:
    rm -rf *.o prog
```

It contains rules describing dependencies and actions with the following format:

- We have the target file on the left, followed by `:`, followed by the file the target depends on. For example `main.o` depends on `main.c`, `network.h and parser.h`; The final executable `prog` depends on `main.o`, `network.o` and `parser.o`; etc.
- Below each target and list of dependencies, we have the command used to rebuild the target. For example `main.o` is created by compiling `main.c`, and `prog` is created by linking the 3 object files. We also have a special rule `all` that will be executed when we type `make` in the local directory. And another special rule `clean` that removes all build file.

After this file is properly set up, typing `make` in the local directory will trigger the rebuild of only what is needed to rebuild the target referenced by `all`.