Lecture 16: Modular 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/16-modular-compilation.

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

Here we discuss modular compilation, the process of decomposing a program's sources into several files and compiling a single executable from these sources.

Modular Compilation: Motivation

With large programs, it is necessary to organise the code in several source files. The Linux kernel v6.4[1] is for example composed of more than 56K C source files. One cannot run `gcc <56K source files>`: not only it is impossible to write manually, but also would take an enormous amount of time to compile each time one do just a small update to one or a few source files. Hence, here we will answer the following question: **how to break up the program code into well isolated compilation units**? In the next lecture we will see **how to automate efficiently the build process?**.

A Closer Look at the Compilation Phase

Until now we saw that when we compile a source file into an executable, set aside the preprocessor phase, the compiler directly transforms the source code into the executable. Actually things are a bit more complicated under the hood. The compiler produces (sometimes transparently) an intermediate format: **object files**. And then we have a second tool named the **linker**, called under the hood by the compiler, that transforms object files into executables.

We can trigger these steps manually with the following commands:

```
$ gcc -c src.c -o src.o
$ gcc src.o -o executable
$ ls
executable src.c src.o
```

First we generate the object file with the `-c` switch when invoking the compiler. The object file here is `src.o`. Then we link it into an executable. With only one source file, the linker does little work, but it plays an important role with modular compilation. The object files are binary files: they contain compiled machine code, however that code is not ready for execution and needs linking. Generally, object files have the `.o` extension.

Breaking Down the Program into Modules

Running Example

Now let us study how to break down the program sources into several source files. It is good to regroup code into source files (named modules in C) by functionality. Let's assume that we have an hypothetical server application

[1] https://github.com/torvalds/linux/tree/v6.4

built from 3 source files:

- `network.c` contains the code related to networking operations.
- `parser.c` contains the code to parse the requests received by the server.
- `main.c` contains the `main` function, initialisation/exit code, and the main server loop.

We can compile each into the corresponding object file as we saw previously, and link all the object files together into an executable as follows:

```
$ gcc -c main.c -o main.o                  # build main.o
$ gcc -c parser.c -o parser.o              # build parser.o
$ gcc -c network.c -o network.o            # build network.o
$ gcc main.o network.o parser.o -o prog    # link executable
```

Application Programming Interfaces

Each source file exposes a set of functions that can be called from other files: an **application programming interface**, API. This interface should be as small as possible to hide internal module code and data. For example if we have a function implemented in `network.c` and this function is only supposed to be called internally within the network module, there is not reason to give `main.c` the *possibility* to call that function.

Defining APIs with Header Files

To define the APIs exposed by each module that can be called externally, we use header files (with `*.h` extension) and the `#include` preprocessor directives to realise this compartmentalization. Let's assume that the external interface offered by the network module is constituted of 2 functions, `init_network` and `receive_request`, both supposed to be called from `main.c`. The interface offered by the parser module is a single function, `parse_request`, also supposed to be called from `main.c`.

We create one header file per module exporting an interface: `network.h` and `parser.h`. These header files will be included in several source files, so it is important that they contain only **declarations**: function prototypes, `struct`/`typedef`/`enum` declarations, variable declarations, etc. They should contain no definitions.

This is `network.h`:

```c
#ifndef NETWORK_H // include guard
#define NETWORK_H

typedef struct {
    int id;
    char content[128];
} request;

void init_network();
int rcv_request(request *r);

#endif /* NETWORK_H */
```

It contains everything that the world outside the network module needs to know in order to use the network interface. We have the prototypes of the two functions of the interface, and also the declaration of the `struct` that is used as parameter in one of these functions. Note the enclosing `#ifdef NETWORK_H`, it's call an include guard. It avoids the problem of double declaration when we include in a C file several files that themselves include this header.

Below is the content of `network.c`, also called its **implementation**:

```c
/* std includes here */
#include "network.h"

// this function and variable are internal
// so they are not declared in network.h
// the keyword static force their use
// to be only within the network.c file
```

```c
static void generate_request(request *r);
static int request_counter = 0;

void init_network() {
    /* init code here ... */
}

int rcv_request(request *r) {
    generate_request(r);
    /* ... */
}

static void generate_request(request *r) {
    /* ... */
}
```

We need to include the corresponding header to get access to the struct definition. We have a function `generate_request` and a global variable `request_counter` that are internal to the module, and they are not supposed to be called/accessed from outside, we can enforce that with the `static` keyword. And we have the implementation of the 2 functions that are exported by the network module.

`parser.h` and `parser.c` follow the same principles:

```c
/* parser.h */

#ifndef PARSER_H
#define PARSER_H

/* needed for the definition of request: */
#include "network.h"

void parse_req(request *r);

#endif
```

```c
/* parser .c */
#include "parser.h"

static void internal1(request *r);
static void internal2(request *r);

void parse_req(request *r) {
    internal1(r);
    internal2(r);
}

static void internal1(request *r) {
    /* ... */
}

static void internal2(request *r) {
    /* ... */
}
```

We have an external interface function declared in the header. We also need the declaration of the `struct` so we include `network.h` in the parser header. And in the module implementation we include the parser header. We have a couple of internal functions, as well as the exported function.

Finally, main.c looks like that:

```c
/* main.c */

#include "network.h"
#include "parser.h"

int main(int argc, char **argv) {
    request req;

    /* call functions from network module */
    init_network();
    rcv_request(&req);

    /* call function from parser module */
    parse_req(&req);
    /* ... */
}
```

It's including both headers to get access to the interface function prototypes. And within the main function the functions from the interface can be called.