Software Development With Emacs: The Edit-Compile-Debug Cycle

Luis Fernandes
Department of Electrical and Computer Engineering
Ryerson Polytechnic University

August 8, 2017

The Emacs editor permits the edit-compile-debug cycle to be easily managed because it integrates very nicely with the `gcc` the GNU compiler, `gdb` the GNU debugger and with revision-control programs like RCS. This close integration unburdens the software developer from the mundane, administrative aspects of coding and allows them to concentrate on writing and developing code.

1 Introduction

This tutorial is an introduction to editing, compiling and debugging C programs with Emacs[1], The One True Editor. After completing this tutorial, you will be know how to:

- write a `Makefile`
- compile C programs with `gcc` from within Emacs
- use `gdb` from within Emacs

This tutorial assumes the reader has a installed the complete Emacs distribution (version 19 or higher) and optionally, RCS (the Revision Control System). It also assumes rudimentary familiarity with Emacs (e.g. knowing how to load and save files) and the C programming language.

To use this tutorial, first read it completely through, once, understand it, and then do it!

Before you begin, create a sub-directory where all the files for the tutorial will be stored and make it your current working directory.

[1]You may use XEmacs if you prefer

2 The Program

We will write a simple program that calculates the mean of up to 10 numbers given a set of numbers read from a file called `data`.

This project will have 2 source-files: `main.c` will contain the function `main()` which will read the data and `mean.c` will contain the function, `calc_mean()` that calculates the mean of the data.

Start Emacs, and enter the following program into a buffer called "main.c":

```
#include <stdio.h>

#define DATAFILE "data" /* name of the data file */
#define MAXDATA 10

int
main(int argc, char **argv)
{
  FILE *fp;
  int count=0;       /* count data items read*/
  int data[MAXDATA]; /* data will be loaded into here */
  float mean;        /* store the mean here */

  if((fp=fopen(DATAFILE,"r"))==NULL)
    {
      fprintf(stderr,"FATAL: Could not open %s\n", DATAFILE);
      exit(-1);
    }

  while(!feof(fp))
    {
      fscanf(fp, "%d\n", &data[count++]);
    }

#if 0
  mean=calc_mean(data, count);
#endif

  exit(0);

}
```

Save "main.c".

3 The data file

Now, create a file called "data" and enter up to 10 integers; each number on a separate line.

```
13
23
5
7
12
```

4 The `Makefile`

Open another buffer called “Makefile” and enter the following (lines beginning with `#` are comments):

```
# The compiler we are using
CC= gcc

# Compilation flags (-g is for debugging information)
CFLAGS= -g

# What the executable will be called
TARGET= mean

# The object file(s)
OBJS= main.o

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
```

Important: Make sure that the last line, (CC)`(CFLAGS) -o` $(TARGET)$`(OBJS)` is indented with a single tab (*do not use spaces*).

Save the “Makefile” buffer.

4.1 The `Makefile` explained

The `Makefile` consists of a set of targets (what you are trying to get accomplished), a set of dependancies and a set of rules on how the target is to be built. The program `make` reads the `Makefile` and builds the target based on the rules and dependencies.

This `Makefile` uses variables to improve readability and configurability. Variable definitions allow the `Makefile` to be used on other projects by just changing the relevant variables; i.e. `TARGET=` and `OBJS=`.

Targets are defined by words which begin in the 1st column and end with a colon. The `all:` target is the default target that `make` looks for when no specific target is specified. Our main target is to build an executable called `mean`. The executable, `mean` (assigned to the variable `TARGET`)), is dependent on the object file `main.o`. The rules to build a target appear just below the target, indented by a TAB.

5 Compiling the Program

Save the buffer `main.c`, and compile using the command `M-x compile RET RET`).

The compilation window should report something like:

```
make -k
gcc -g -c main.c
gcc -g -o mean main.o

Compilation finished at Wed May 25 14:30:38
```

6 Debugging with gdb from within Emacs

We are now going to use the debugger to step through the program statement by statement and view the contents of various variables our program.

To run the debugger, type:

```
M-x gdb RET test RET
```

CHECK WHETHER THE WINDOW SPLITS HERE OR LATER!!!

The window will split and a gdb-buffer will appear with the `(gdb)` prompt.

6.1 The break command

Type: `break main`, at the prompt. (The `break` command tells `gdb` to place a "breakpoint" at the start of the function called `main`; a program stops execution at every breakpoint.)

`gdb` will respond with something like:

```
Breakpoint 1, main (argc=1, argv=0xdffff904) at main.c:12
```

6.2 The run command

Now type: `run`. The `run` command instructs `gdb` to begin executing the program from the beginning. Any command-line arguments typically passed to a program can be typed after the `run` command.

Your Emacs window will split, to display a gdb window at the top and the source-code at the bottom CHECK THIS!!!. The source-code window will display an arrow `=>` in the left-margin, indicating to the next line to be executed.

6.3 The step and print commands

Type: `s`, in the gdb window to execute the statement (`s` is an abbreviation for "single-step").

Type: `s`, again to execute the next line; if the data-file does not exist, then typing `s` twice more will complete the execution of the `if` statement with the error-message being printed out.

(Create the data-file now, save it and re-run the test-program from within gdb.)

At this point, the data-file has been opened for reading and gdb should be pointing that the start of the "while"-loop.

Stepping twice will execute the "while" and then the "fscanf". At this point we can check whether the first datum has been read correctly.

Type: "p data[0]". gdb will print the contents of data[0].

Typing "s 3" will execute "step" 3 times; "p data[1]" will print the contents of the next datum.

Typing "c" will continue executing until the next breakpoint. Since we do not have another breakpoint, the program will execute to completion.

Next, we'll code the function to calculate the mean.

7 calc_mean()

Create a new buffer called `mean.c` and type-in the following code:

```
/* This function calculates the mean of a set of data and returns it
 * on the stack, passed a pointer to the data-array and the number of
 * items in the array.*/

float
calc_mean(int *data, int count)
{
  register int i;
  float mean=0;

  for(i=0; i<count; i++) mean+=data[i];

  return(mean/i);
}
```

Next, we have modify `main.c` as follows:

First we add a forward-declaration[2] of our new function:

```
float calc_mean(int *data, int count);
```

Finally we uncomment the function-call to `calc_mean`, just after the tt while-loop:

```
mean=calc_mean(data, count);
```

[2]The forward-declaration tells the compiler that the function `calc_mean` will be passed a pointer to an array (containing the data) and an integer (number of valid data in the array) and that it will return a floating-point number (representing the mean).

`main.c` now looks like this:

...

We also update the `Makefile` to let it know of our new source-file `mean.c` which will be compiled to the object file `mean.o`:

```
OBJS = main.o mean.o
```

After saving the `Makefile`, re-build the program by typing `make` in the `gdb` buffer; the output will be something like:

```
gcc -g -c main.c
gcc -g -c mean.c
gcc -g -o mean main.o mean.o
```

Both `main.c` and `mean.c` are compiled because they were *both* modified.

Note that `M-x compile RET RET` would have worked just as well as typing make in the `gdb` buffer. It would also have allowed integrated access to any errors, via `` C-x ` ``.

8 Debugging `calc_mean()`

Let's re-run the program to check whether the new code we have added works.

We can add a break-point in the function that calculates the mean by typing `break calc_mean` in the `gdb` window; `gdb` will print something like:

```
Breakpoint 2 at 0x10900: file mean.c, line 5.
```

indicating that break-point 2 is set in the file containing that function (recall that break-point 1 is set at `main()`).

Type `run` in the `gdb` window to begin running the program under the debugger.

8.1 The `continue` command

The debugger will stop the program at `main()`.

Next, type `c` (abbreviation of "continue") to continue execution until the next break-point. The execution pointer `=>` will stop in `calc_mean` at line number 5.

Typing `s` will execute the line that zeroes the variable `mean`; another `s` will execute the `for`-loop which sums all the data.

Typing `p mean` will display the contents of the sum stored in mean (this value can be verified with a calulator).

Stepping twice more will execute the actual calculation of the mean by dividing by the number of values that were summed.

Once the execution returns to `main()` we can print the contents of `mean` to verify that the division was performed correctly by typing: `p mean`.

The program finishes by executing `exit()`. `gdb` reports: `Program exited normally.`