CS 305 Lab 4: make and the debugger
Fall 2019

This lab introduces you to some of the basics of using the utility `make` and using the command-line debugger `gdb`. This lab has a total of 100 possible points (30 points from pre-lab and 70 points from lab).

Sit with your assigned partner and switch the roles of driver and navigator throughout the lab.

Objectives

Upon completion of the laboratory exercise, you will be able to do the following:

- Create a program containing multiple source files; and compile the source files and link the object files.
- Understand the dependency tree for various compilation modules.
- Write, debug, and test a makefile for selective compilation using the `make` utility.
- Use gdb to locate and fix bugs in a C program; set breakpoints; step through code; and inspect the stack.

Part 1: Logging in

1. Follow the procedure from prior labs to login, open Mobaxterm, and navigate to your cs305 directory. Create a new lab4 directory:

```
cd cs305
mkdir lab4
```

2. Download the `fact.c` file to your lab4 directory.

Part 2: Getting files from prelab

1. Create a subfolder in your lab4 folder called part1.

```
cd lab4
mkdir part1
cd part1
```

Using the partner who logged in, copy the files from their prelab (makefile and test1.c) to this folder. You can do this with the file explorer in Windows or the `cp` Unix command to copy a file.

```
cp ../../prelab4/test1.c .
cp ../../prelab4/makefile .
```

(This assumes your directory structure is such that cs305 -> prelab4 and cs305 -> lab4 -> part1.)

2. Open the makefile to be sure it has the following lines at the bottom of the file:

```
# this defines the command for cleaning up
# all generated files
clean:
	/bin/rm -f *.o test1
```

(where the character before the `/bin/rm` is a single TAB character).

3. Close the makefile and type:

```
make clean
```

Part 3: Modifying the sources (switch driver)

1. Use emacs to open a new file called `test1.h`. Move the definition of the constant (macro) `SIZE` from `test1.c` to `test1.h`. (That is, copy the line into `test1.h` and delete it from `test1.c`.)

2. Add the line

```
#include "test1.h"
```

after the other `#include`-line in `test1.c`. Don’t forget to place quotations around the file name.

Declarations that `test1.c` wants to make visible to other modules are put in its ".h" or *header* files. Using header files is a standard programming style for the C language. As a result, the entire program consists of a collection of files that may be independently written, debugged, compiled, and linked together. This is good for modular program design and for having multiple people work on the same code base. In our example,

- `test1.c` has `#include "test1.h"` to ensure that external definitions in the ".h" file are consistent with internal definitions in the ".c" file.
- Other files that need to "include" `test1.h` do so by adding `#include "test1.h"`. Then later, the linker combines the respective ".o" files into a single executable file.

3. Type `make test1`. Because you had just done a `make clean`, this should rebuild the executable file `test1` from it sources again. Does it? (circle answer)
YES / NO

Does the program `test1` run just as it did before? (circle answer) YES / NO

4. Modify `test1.h` so that `SIZE` has the value 5; rebuild (i.e., type `make test1`). Does the execution of `test1` reflect the change just made to `test1.h`? YES / NO

If so, explain why. If not, tell what change(s) must be made to the `makefile` so that it prints the correct value, make those changes, and confirm that they work.

5. Add the call

```
print_president();
```

to `main` in `test1.c` just before the return statement.

Also, add the line

```
#include "president.h"
```

to the list of #includes.

6. Create a file named `president.c` that defines the function `print_president` as follows:

```
#include <stdio.h>
#include "president.h"

void print_president() {
      printf("My favorite president in history is xxx.\n");
}
```

where you replace "*xxx*" with the name of your favorite president.

7. Create the file named `president.h` to have just the prototype declaration:

```
void print_president();
```

8. Things are now set up in the "standard" way:

- the declaration of `print_president` (which is just the function header) is in the ".h" file
- the definition of `print_president` (which includes the body) in the ".c" file.

9. If you type `make test1` now, you will get some errors. Make the modifications to the `makefile` so that all build-commands are correct, and so that all dependencies are properly specified. You will need to add a line for `president.o` and update the lines for `test1` and `test1.o`. Draw the dependency graph for the various compilation-units below:

Part 4: Make variables (switch driver)

1. Filenames, commands, switches, etc., can be specified symbolically. These symbols can be defined in three places:

- On the command-line when `make` is invoked, as in `make myCount=50 test1`.
- In the `makefile` itself on a line such as `myCount = 50`.
- Taken from a shell environment variable. A shell environment variable is set by giving a command such as `export someVar='someValue'`.

2. We will be modifying the `makefile` so first **save** your old one as `makefile1`. You can do this easily with the `cp` command:

```
cp makefile makefile1
```

3. A `make` variable such as `myVar` is referenced in the `makefile` using the (rather awkward) syntax

```
$(myVar)
```

4. In order to better parameterize the `makefile`, add the lines

```
# define the compiler-executable path
COMPILE = gcc
```

to the beginning of `makefile`, and replace all invocations of `gcc` with `$(COMPILE)`. Perform a `make clean` and a `make test1` to ensure that things still build. Do they?

YES / NO

5. Try the following commands

```
make clean
make COMPILE='gcc -g' test1
```

6. How does the behavior differ from that of the previous "make"?

7. Use the Unix/bash `export` command to set the environment-variable `COMPILE`. Type the following into the shell prompt (it’s a big O (oh), not the number zero in the command below):

```
export COMPILE='gcc –g –O4'
```

8. Now do a `make clean` and a `make test1`. Then, comment out the definition of `COMPILE` in the `makefile` and try it again. What does it tell you about the priority that make gives to each of the following?

- Value specified on command line. Priority (highest = 1, lowest = 3):
  1 2 3
- Value read from environment variable (export). 1 2 3
- Variable’s value specified in the `makefile`. 1 2 3

9. Uncomment the definition of `COMPILE` in the `makefile`. Append the token `$(COMPILE_SWITCHES)` to the definition, as in

```
COMPILE = gcc $(COMPILE_SWITCHES)
```

10. Perform the following commands

```
make clean
make test1
```

and

```
make clean
make COMPILE_SWITCHES='-g -DLENGTH=4' test1
```

11. How is the definition of `COMPILE_SWITCHES` reflected in the build?

12. Another place where variables are used is in specifying the set of '.o' files that link together to form an executable, as in

```
OFILES = myExec.o helper.o util.o
 ...
myExec: $(OFILES)
     gcc -o myExec $(OFILES)
```

In this way, the set of object files is specified in one place rather than in two, as in:

```
myExec: myExec.o helper.o util.o
     gcc -o myExec myExec.o helper.o util.o
```

Using a variable helps keep the dependencies consistent with the actual object files that are used to build the executable.

13. Change your `makefile` so that all of the '.o' files are defined with a make variable called `OFILES`.

```
OFILES = test1.o president.o
```

```
test1: $(OFILES)
      $(COMPILE) -o test1 $(OFILES)
```

Then, do a `make clean` and then a `make test1` to ensure that it still builds properly.

Part 5: FYI: Pattern-rules and built-in rules (switch driver)

1. Make allows rules to contain wild-cards, using % characters in the pattern and the $@ and $< tokens in the commands. This can be used to specify something like a ".o file is created from a .c file by compiling it with the -c switch", as follows:

```
%.o: %.c
        gcc -c $<
```

The `%` represents a character string that must match in every occurrence in the pattern. The `$<` represents the first filename after the ':' token. A `$@` represents the name of the target, thus the rule

```
%.run: %.c
        gcc -o $@ $<
```

will cause make to execute the command `myProg.run` if `myProg.c` exists:

```
gcc -o myProg.run myProg.c
```

The `make` program has a number of built-in rules, mostly of the "pattern" variety above, that obviate the need for putting explicit rules in straightforward cases.

Checkpoint 1 [30 points]: Show your lab instructor/assistant your answers to the above questions; let your instructor/assistant examine makefile1 and makefile; show your instructor/assistant the results of doing a "make test1".

You may read more about make with `make -h`. The second part of this lab will focus on using `gdb`, a command-line debugger.

Part 6: Compiling with the debug option

1. Go back to your `lab4` directory. Open the file `fact.c` in `emacs`. Examine the code and see if you can spot one or more bugs in the program. Even if you find the bug now, you should continue the lab and use the debugger. Note that since this factorial program uses recursion, only values up to 12 should be used. After 12, the program will take too long to run.

2. If you need to refer to documentation on gdb during the lab, here is a website: http://sourceware.org/gdb/current/onlinedocs/gdb/

3. Compile `fact.c`:

```
gcc -g -o fact fact.c
```

The `-g` switch used during compilation tells the compiler to create debugging information which is necessary in order to run `gdb`.

4. Run the compiled program.

```
./fact
```

When prompted for input, type 2. What does the program print for `2!`? _______

5. Re-run the program. When prompted for input, type 5. What does the program print for `5!`? _______

6. What should `2!` be? ___________ What should `5!` be? ____________

Clearly, the program is not calculating the correct value.

Part 7: Run the program with gdb (switch driver)

1. To debug the program whose executable file is `fact`, use the `gdb` command and specify `fact` as the argument.

```
gdb fact
```

You should see a list of copyright information and a prompt that starts with (gdb). This command has gdb read the debugging information for fact and sets gdb up for running the fact program.

2. To run the program under gdb, use the run command. This command terminates any existing processes and creates a new process that is run with the specified command line arguments. In gdb, type:

```
(gdb) run
```

3. Type 5 when you are prompted for a number.

Part 8: The list, break, print, and cont commands

1. Note: if you type the “up” key, gdb recalls the most recent command typed.

2. Use gdb’s list command to display the lines near the beginning of the main function:

```
(gdb) list main
```

Which line numbers are shown to you? __________

3. Type list again to show the next few lines:

```
(gdb) list
```

4. Notice that line 23 reads the value of `n` from standard input, and that line 27 checks if the value of `n` is less than 0 or greater than 12. Perhaps the program doesn't receive the correct value of `n`. So let’s check the value of `n` using the break, run, and print commands:

```
(gdb) break 27
```

5. The break 27 command directs gdb to set a breakpoint just before executing the machine code that corresponds to line 27 in the program. Note that gdb echoes the breakpoint settings.

6. Now run the program and print the value of `n` at the breakpoint.

```
(gdb) run
...
(gdb) print n
```

7. What is the value of `n`? ___________

8. The program has the correct value of `n`, so the problem must be elsewhere. Use the 'cont' command to continue execution of the process after the breakpoint.

```
(gdb) cont
```

Once again the final printing result is incorrect. Since the program is printing the correct value of n, and only the line containing the call to the `factorial` function remains, the problem must reside in `factorial`.

Checkpoint 2 [10 points]: Demonstrate to your lab instructor/assistant that you can run to the breakpoint, and that you can continue from it to the end of the program. If you need to wait for the checkpoint, take a screenshot now and move on. When the assistant or instructor can get to you, show him/her your screenshot.

Part 9: More breakpoints (switch driver)

1. Try to call the function by itself:

```
(gdb) print factorial(5)
```

2. What happens? ____

3. When told to print the value returned by a function in a program, gdb actually executes the code in the function, passing it the specified value(s) as parameters. Since the current process has exited, gdb cannot execute any code. To run the `factorial` function, you need to create a new process. Do this by setting a breakpoint at `main`, typing `run`, and then `print factorial(5)`.

```
(gdb) break main
(gdb) run
(gdb) print factorial(5)
```

4. We still have not found the source of our trouble, let's keep hunting. Again let's use the 'list' command to list the factorial function.

```
(gdb) list 36,49
```

Let's take a closer look at the `factorial` function. But, first, use the `info break` and `delete` commands to display and delete all unnecessary breakpoints.

```
(gdb) info break
```

5. What information does this command display (shorthand is ok)?

6. Let’s delete the breakpoints. Type:

```
(gdb) delete 1
(gdb) delete 2
```

You typed `delete 1` above because the `info` command reported the (1), indicating that "name" of the breakpoint at line 25 is '1'. (Similarly for breakpoint 2.)

7. Set breakpoints at lines **38** (the first line of function factorial), **40** and **44** (the two return statements).
Then, run the program from the beginning. You may be asked to run from beginning; if so, say y.

```
(gdb) run
```

You should see something like:

```
Breakpoint 3, factorial (n=5) at fact.c:38
38    if ((n = 1) || (n = 0)) {
(gdb)
```

The factorial (n=5) at fact.C:38 is telling you that you are about to execute line 38 of the function factorial and the parameter `n` has value 5. In particular, that means the parameter value is correctly passed between `main` and `factorial`. Resume executing the program.

```
(gdb) cont
```

You should see something like:

```
Breakpoint 4, factorial (n=1) at fact.c:40
40    return 1;
(gdb)
```

How strange! Now you're about to return the bogus result `1` and it appears the value of `n` has changed to `1`. Print `n` to confirm:

```
(gdb) print n
```

8. Run the program to completion.

Part 10: Next and step commands

Like other debuggers you may have used before, gdb has `next` and `step` commands. Both commands execute one line of code. The difference is that "step" will step into a function call if one or more exist on the line; "next", on the other hand treats all function calls as part of the code executed on this line, and stays in the current function. (Exceptions: if a break point is hit, or if a return is being done from the function.)

1. Clear all break points; then set one in the **first statement** in `main`. Find the line number of the first statement in `main` to do this. If you have a program that has multiple source (.c) files, you can specify the name of the source file, a colon, and a line number like this. Let’s do this, so you know how to debug programs with multiple files.

```
break fact.c:17
```

Use a combination of `next` and `step` commands (or their abbreviations "n" and "s") to get to the first statement in the `factorial` function.

2. When you reach the first statement in `factorial`, print the value of `n`; then alternatively give "next" and "print n" commands to see which statement caused `n` to prematurely change.

3. What is the buggy line of code?

Part 11: Fix the bug (switch driver)

1. Hit q to exit gdb. Open `fact.c` in emacs. Carefully examine the offending line of code. Fix the problem by editing `fact.c`. Save it. Then recompile the program, making sure you use the `-g` switch.

2. Now run the program. Hopefully, the correct value for the factorial will be printed. If not, keep debugging.

Checkpoint 3 [20 points]: Show your lab instructor/assistant that the fact program is now working correctly. Show your lab instructor/assistant the answers to the questions above. If you need to wait for the checkpoint, take a screenshot of your program working.

Part 12: Examine the stack (switch driver)

1. Rerun `gdb` on the executable and set a breakpoint at the line with the function definition for `factorial`. Then run the program, using `5` as the input.

2. Give the `step` (or "s") command at the next seven prompts. What is the value of `n` at this point?______

3. We have dropped into several nestings of function calls using `step`. We can see all the active function-calls using the command:

```
(gdb) info stack
```

4. Run the program from the beginning using `5` as input. Use the `info stack` command each time the breakpoint is hit, use `c` to continue running until the breakpoint is hit again, and take note as to how the stack grows.

At its largest point, how tall is the stack?________________

5. Print the value of `n` when the stack is at its largest. What is the value of n?_______
Use the `up` command. Now print the value of `n` again. What is the value of `n`? ______

6. Are the two values of `n` above different?_________

Explain why they should (or should not be) different (use the `help up` command, if you'd like):

Quit the debugger.

Checkpoint 4 [10 points]: Show your answers to the above questions to your lab instructor/assistant.

FYI: A Visual debugger

There is a visual debugger called `ddd` that you can install as a Mobaxterm package. If you like to set breakpoints with a visual tool, you are welcome to use this. Type `ddd` in Mobaxterm and open your executable. The graphics/menus are straightforward. Try debugging the factorial program using this visual debugger.

Now that you know how to use the debugger, I expect you to use this to debug your code with gdb or ddd. It is helpful to set breakpoints when debugging seg faults. Set a breakpoint in main, then step as far as you can go until the program crashes.

Close Mobaxterm and any other applications. Log off the computer.

If any checkpoints are not finished, they are due in one week. You may submit screenshots, code files, and answers to questions via printouts or email.