University of Windsor

School of Computer Science
Faculty of Science
COMP-2560: System Programming (Fall 2022)

Lab#	Date	Title	Due Date	Grade Release Date
Lab04	Week 04	**Bash Scripts for Test Cases**	Oct. 19, 2022, Wednesday Midnight EDT	Oct. 24, 2022

The main objective of this lab will be to learn how to use shell scripts for the benefit of programmers. *By no means, this lab or this course is enough to cover every aspect of shell scripting (writing scripts for shells.)* We will only shed light on some aspects that help programmers and showcase the powers. Shell scripting follows the same principles as C programming language does. Hence, if you know C programming, it will be straightforward to master shell scripting. Can shell scripting be a substitute for C programming?

Step 1. Simple `bash` Script

In Lab03, we saw that we have to manually run C compiler, `cc`, for any change to our lines of C program, with different options. Some shells, *not all*, have been programmed to accept a sequence of requests in one single file, called *scripts*. Whatever you can request from a shell, like bootstrap a program, run a built-in command, etc., can be put in a file (script). Then, you can ask the shell to run the lines of your file (scripts) sequentially. Let's have a look at the very simple example to improve building the program from Lab03. *Pleases note that this lab does not depend on Lab03.*

The main file accepts an input value from the user and outputs the final result:

```
hfani@alpha:~$ vi main.c
```

```c
#include <stdio.h>
int increment(int a);
void main(void){
        int a;
        scanf("%i", &a);
        a = increment(a);
        printf("%i \n", a);
        return;
}
```

Next is the file that includes the C lines for incrementing a given number:

```
hfani@alpha:~$ vi increment.c
```

```c
int increment(int a){
        int b = a;
        b = b + 1;
        return b;
}
```

Both files should be compiled into assembly codes:

```
hfani@alpha:~$ cc main.c -S
hfani@alpha:~$ cc increment.c -S
hfani@alpha:~$ ls
Desktop  Documents  Downloads  eclipse-workspace  increment.c  increment.s  main.c  main.s  Mus
```

Followed by translation to opcodes (object files) by the assembler:

```
hfani@alpha:~$ cc main.s -c
hfani@alpha:~$ cc increment.s -c
hfani@alpha:~$ ls
Desktop    Downloads          increment.c  increment.s  main.o  Music     Public
Documents  eclipse-workspace  increment.o  main.c       main.s  Pictures  Templates
```

Finally, we have to link (statically) the two object files into a single executable file:

```
hfani@alpha:~$ cc main.o increment.o -o main
hfani@alpha:~$ ./main
3
4
hfani@alpha:~$
```

If you want to change some lines of `main.c` or `increment.c`, you have to **re-run** all these steps manually:

```
hfani@alpha:~$ cc main.c -S
hfani@alpha:~$ cc increment.c -S
hfani@alpha:~$ cc main.s -c
hfani@alpha:~$ cc increment.s -c
hfani@alpha:~$ cc main.o increment.o -o main
hfani@alpha:~$
```

A better way would be to put all these requests from the shell inside a script file:

```
hfani@alpha:~$ vi build_main.sh
```

```
#!/bin/bash
echo "start building main program:"
echo "compiling to assembly lines ..."
cc main.c -S
cc increment.c -S
echo "translating to opcodes ..."
cc main.s -c
cc increment.s -c
echo "statically linking all required opcodes ..."
cc main.o increment.o -o main
echo "build successfully done!"
```

Please note the following:

- The filename for scripts has `.sh` extension by convention (it could be anything, though.).
- The first line shows that this is a script for bash shell (`#!/bin/bash`). This is important because a script for a shell may not be supported by another shell. *Note that a shell may not support scripting at all! Like what?*

To run a script, there are two ways: either 1) explicitly give it to the shell,

```
hfani@alpha:~$ bash build_main.sh
start building main program:
compiling to assembly lines ...
main.c: In function 'main':
main.c:5:6: warning: implicit declaration of function 'increment' [-Wimplicit-function-declaration]
    5 |  a = increment(a);
      |      ^~~~~~~~~
translating to opcodes ...
statically linking all required opcodes ...
build successfully done!
hfani@alpha:~$
```

Or 2) run it as a program. This way, you have to make the script file executable by the built-in command `chmod` (change modes.) This command changes the attributes of a file to help the shell and kernel about the content.

```
hfani@alpha:~$ chmod +x build_main.sh
```

Then, you can run the script file as an executable program:

```
hfani@alpha:~$ ./build_main.sh
start building main program:
compiling to assembly lines ...
main.c: In function 'main':
main.c:5:6: warning: implicit declaration of function 'increment' [-Wimplicit-function-declaration]
    5 |  a = increment(a);
      |      ^~~~~~~~~
translating to opcodes ...
statically linking all required opcodes ...
build successfully done!
hfani@alpha:~$
```

Now, any change to the `main.c` or `increment.c` can be built quickly by running the script file. Let's test our program for an input value `4`:

```
hfani@alpha:~$ ./main
4
5
hfani@alpha:~$
```

Step 2. `bash` Script for Test Cases

The next step would be to include bootstrapping our program inside the script:

```
echo "running the main program"
./main
```

If you run this script, it will run your program if it's built successfully in the previous steps:

```
hfani@alpha:~$ bash build_main.sh
start building main program:
compiling to assembly lines ...
main.c: In function 'main':
main.c:5:6: warning: implicit declaration of function 'increment' [-Wimplicit-function-declaration]
    5 |  a = increment(a);
      |      ^~~~~~~~~
translating to opcodes ...
statically linking all required opcodes ...
build successfully done!
running the main program
```

So, we can call this script a *test* script for our program. Let's make a copy of it with a new name:

```
hfani@alpha:~$ cp build_main.sh test_main.sh
```

We can include the input values inside the script to thoroughly test our program and check the result. This can be done by `<<<` operator, called *here-string.*

```
#echo "running the main program"
#./main
echo "running the main for input 4:"
./main <<< 4
echo "running the main for input 10:"
./main <<< 10
```

As seen, our program is going through two runs for different input values `4` and `10`. We expect that our program outputs correct results for both:

```
hfani@alpha:~$ bash test_main.sh
start building main program:
compiling to assembly lines ...
main.c: In function 'main':
main.c:5:6: warning: implicit declaration of function 'increment' [-Wimplicit-function-declaration]
    5 |  a = increment(a);
      |      ^~~~~~~~~
translating to opcodes ...
statically linking all required opcodes ...
build successfully done!
running the main for input 4:
5
running the main for input 10:
11
hfani@alpha:~$
```

It seems our program could successfully *pass* these two tests. In software engineering, a test instance is called a *test case* under which a program may be *passed* or *failed*.

Step 3. Advanced `bash` Scripts

Any input value can be expected from a random user in practice. What if the user inputs a real number `1.0`? How about a number but in chars `"1.0"`? We have to make sure that our program is robust, even in rare situations. One way is to add lines of test cases in our test script `test_main.sh`. But it is very inconvenient.

How about split the duties. Put the input values in another file. The script `test_main.sh` opens the file, fetches a new input value at each time, runs our program for that value, then continues to the next input value, and so on. Is it possible to read a file in a shell script? Is it possible to have a loop? Yes!

Let's create a text file and put all random input values to our programs as test cases. As seen, we separate different input values to our program by comma `,`

```
hfani@alpha:~$ vi test_inputs.txt
```

```
1
-20
37
1.4
0
0.0
'20'
lab04
```

Now, we have to open `test_inputs.txt` in `test_main.sh` and read the content to the end. This can be done by the following syntax:

```
while read variable
do
     {something}
done < "{filename}"
```

Basically, the shell opens `{filename}` by `<` for reading. Then, it reads a line by `read` and puts the whole line into `variable`. The shell continues until there is no line. Let's do it for our purpose:

```
while read input
do
        echo "test the main program for $input"
        ./main <<< $input
done < "./test_inputs.txt"
```

Here, our variable to hold a line is `input`, and the filename is `test_inputs.txt` in the current directory `./`. You can see that when we refer to a variable, we put a `$` sign like in `$input`. Let's run the script:

```
hfani@alpha:~$ bash test_main.sh
start building main program:
compiling to assembly lines ...
main.c: In function 'main':
main.c:5:6: warning: implicit declaration of function 'increment' [-Wimplicit-function-declaration]
    5 |  a = increment(a);
      |      ^~~~~~~~~
translating to opcodes ...
statically linking all required opcodes ...
build successfully done!
test the main program for 1
2
test the main program for -20
-19
test the main program for 37
38
test the main program for 1.4
2
test the main program for 0
1
test the main program for 0.0
1
test the main program for '20'
1
test the main program for lab04
1
hfani@alpha:~$
```

As seen, the scripts open the files and run the `main` program for each line of it.

However, it seems our program failed for some test cases when input values are `1.4`, `'20'`, `lab04`! Specifically, we have 4 passed and 4 failed test cases (`0.0` should become `1.0`).

Step 5. Lab Assignment

You should write a script that outputs the number of passed and failed test cases. To this end, you need to have the correct answer in `test_inputs.txt` as well:

```
1,2
-20,-19
37,38
1.4,2.4
0,1
0.0,1.0
'20','21'
lab04,lab05
~
~
```

At each iteration, you have to 1) run `./main`, 2) give it an input value, 3) store the output of the program in a new variable, 4) compare the value of this variable with the correct output, 5) count up the passed if they are equal and count up the failed otherwise, 6) shows the (input value, program output value, correct output value) pairs like the following:

```
input: 1, main: 2, correct: 2 ==> passed
input: -20, main: -19, correct: -19 ==> passed
input: 37, main: 38, correct: 38 ==> passed
input: 1.4, main: 2, correct: 2.4 ==> failed
input: 0, main: 1, correct: 1 ==> passed
input: 0.0, main: 1, correct: 1.0 ==> faild
input: '20', main: 1, correct: '21' ==> failed
input: lab04, main: 1, correct: lab05 ==> failed

total passed: 4
total faild: 4
```

The sample code for steps 1 and 2 has been attached in a zip file named `lab04_hfani.zip`.

1.1. Deliverables

You will prepare and submit the program in one single zip file `lab04_uwinid.zip` containing the following items:

(90%) `lab04_uwinid.zip`

- (05%) `main.c` => built with no error
- (05%) `increment.c` => built with no error
- (10%) `test_inputs.txt` => contain input values and correct output values
- (60%) `test_main.sh`
  - (10%) successfully build the main program
  - (10%) apply all test cases to the main program
  - (40%) correctly count passed and failed test cases.
- (10%) `results.pdf/jpg/png` => the image snapshot of the script output
- (Optional) `readme.txt`

(10%) Files Naming and Formats

Please follow the naming convention as you lose marks otherwise. Instead of uwinid, use your own account name, e.g., mine is hfani@uwindsor.ca, so, `lab04_hfani.zip`