Faculty of Computer Science, Dalhousie University *4-Oct–2018*

CSCI 2132 — Software Development

Lab 4: Exploring bash and C Compilation

Location: Teaching Labs Instructor: Vlado Keselj
Time: Thursday

Lab 4: Exploring bash and C Compilation

Lab Overview

- Exploring shell (bash)
- Compiling C programs

In this lab, you will first learn a little more about the bash shell and how to set it up. Then, you will use the Unix `diff` command to compare two files. You will then write and compile some C programs using `emacs` and `gcc`. In this process, you will also get some experience in using the job control mechanism of the Bash shell.

Be sure to get help from teaching assistants whenever you have any questions.

Step 1: Login and Lab Setup. Login to your bluenose account as before. Go to your checked out copy of your course SVN repository. Based on the previous suggestions, it should probably be in the following directory: `~/csci2132/svn/`*CSID* where *CSID* is your CSID.

Create and add `lab4` directory to the SVN. Commit the change to SVN. Change your current directory to `lab4`. You will work in this directory.

Step 2: Exploring shell (bash) Let us first verify which shell we are running; type:
`echo $SHELL`
This should produce the output '`/bin/bash`', so we know that we are indeed running the bash shell. You can see what other shells are available by typing:
`cat /etc/shells`
The output should be a list of available shells, including '`/bin/bash`', and one, actually, non-shell program '`/sbin/nologin`'. After this short exercise, let us move to the next step, were we will see how to set up some default shell behaviour.

Step 3: `.bashrc` file. The file `.bashrc` is usually used to set up the initial commands for the bash shell, which are executed every time we login into the system. As a result of this exercise, you will edit this file, or at least a copy of this file.

Important Warning: You need to be careful to execute steps as described, since invalid `.bashrc` file can cause that you cannot login to your account, and you will need help from Help Desk to fix this problem.

One thing which we usually want to change is behaviour of the `rm` command. When deleting a file, as in the command '`rm file1`', the `rm` command immediately removes the file without asking for confirmation. This is a risky behaviour, which may lead to accidental removal of a useful file. We can provide the option `-i` to `rm`, but then we need to type this option every time we use the command `rm`. A way to avoid this is to use the bash built-in command `alias` to override command `rm` with the command '`rm -i`', and the command `alias` should be in `.bashrc` file so that we do not have to execute it every time. First, let us check if `rm` is already an alias, type:

`which rm`
The output should be '`/bin/rm`'.

Copy your `~/.bashrc` file into your `lab4` directory into the file named `bashrc.old`. If the file `~/.bashrc` does not exist, then create an empty file `bashrc.old`. You can create an empty file by using the '`touch`' command. Add the file `bashrc.old` to SVN.

Copy the file `bashrc.old` to the file `bashrc.new` and add the `bashrc.new` file to SVN.

We will now edit the file `bashrc.new` using emacs:

Enter the following contents into the `bashrc.new` file (if there is some content already there you can add these lines at the bottom of the file):

```
umask 077
alias rm="rm -i"
alias mv="mv -i"
alias cp="cp -i"
```

Warning: You should be careful to copy the contents of the file exactly as shown. This file will be copied to the original `.bashrc` file, and you should remember that it is very important that this file is valid for the login process.

These are all commands which can be entered in the command line as well. The following is a short description of what they do:

- `umask 077`: sets the user file-creation permission mask for the shell. When new files are created, the shell uses this mask to determine what should be initial file permissions. For example, a mask 077, as above, would make by default any new file readable, writable, and executable, if applicable, to the user, but not to the group or others (0 is for the user, 7 for group, and 7 for others). Remember that octal numbers are used for the masks.
- `alias rm="rm -i"` sets an alias `rm` to be replaced with `rm -i`, where `rm` refers to the command. So, each time we want to remove the file, we will be asked for a confirmation. The aliases defined for `mv` and `cp` are very similar. If a move or copy operation is going to overwrite another file, we will be asked for a confirmation.

Verify that the file `bashrc.new` is valid by running the command:
`source bashrc.new`
There should be no errors reported. You should check once more that you carefully copied the given contents to this file. You can also check with `which rm` that the command `rm` has the proper alias—the output should be:

```
alias rm='rm -i'
/bin/rm
```

Do not forget to add both files `bashrc.old` and `bashrc.new` to SVN. You can do SVN commit at this point too.

Optional step: Now, you can copy the new file `bashrc.new` to `~/.bashrc`. This step will not be marked and you do not have to do it. This is a **"dangerous"** step, since invalid `~/.bashrc` may break your login ability. If you are not confident about it, you can skip it. If you want to proceed, you should follow the following guidelines: After copying the file, you should test it. To test the file, you should open another terminal window (with putty or ssh) and try to login to see that everything is okay. If your login is broken, you should go back to the first terminal window and copy your `bashrc.old` file to `~/.bashrc` and then try to fix the problem in `bashrc.new`.

If your login in another window is successful, you can check in it if the behaviour of the `rm` command has changed by typing:
`which rm`

You will probably get the output '/bin/rm', which means that the alias is still not taking effect. Actually, .bashrc is not executed by default. The default file that bash executes each time we login is .profile, but since this file is used by several other shells, we usually just put a shell script there to call an appropriate other starting script.

Step 4: Editing .profile File. Similarly as with ~/.bashrc file, you should copy ~/.profile file to profile.old in your lab4 directory, and if ~/.profile does not exist then create an empty profile.old using the command touch. Add profile.old to SVN. Copy profile.old to profile.new and add profile.new to SVN.

Using emacs or some other editor, you should edit profile.new so that it has the following contents. It is possible that the file contains already these or similar lines, in which case you can leave it as it is.

```
case `basename $SHELL` in
  sh|jsh)
        . $HOME/.shrc
        ;;
  ksh)
        . $HOME/.kshrc
        ;;
  bash)
        . $HOME/.bashrc
        ;;
esac
```

Warning: You have to type the contents exactly as shown. In particular, be sure to use **backquotes** (`) in the string `basename $SHELL` above. This is a frequent error. We will learn more about backquotes in shell later, but a brief explanation for now is that a string between backquotes is interpreted by a shell as a command, which is executed and the result is used as a replacement string for the command.

Even though we did not start learning shell scripts yet, you should be able to understand the logic behind the above script. It is a branching statement, similar to switch in Java, which, based on the basename of the shell, calls different starting scripts. In our case, the shell is bash, so .bashrc file in our home directory is called.

Warning 2: Again, be sure to enter the contents of profile.new file exactly as shown. Otherwise, you may not be able to enter into your account once this files is copied to ~/.profile.

Check the new file by running the command: source profile.new

There should be no errors.

It is possible that this command will report an error that '~/.bashrc' does not exist ('No such file or directory'). This happens because profile.new is trying to run this file and it does not exist. This should not be a concern, but if you want to be sure that this is the only problem, you can fix it using the command:

```
touch ~/.bashrc
```

and after that the command 'source profile.new' should report no errors.

Optional step: This is an optional step, and again you do not have to do it if you do not feel confident about updating your '.profile' file. Remember that if this file is not valid, there could be problems logging in into your account.

If there are no errors, copy the file profile.new to ~/.profile and use login through another terminal window to check that everything is okay. If there are problems logging in, go to the first window and copy the file profile.old to ~/.profile and fix the problem.

SVN Submission: Commit the files profile.old and profile.new to SVN.

Hopefully, everything is okay by this step. You can now check with '`which rm`' in the second terminal window that the command `rm` is properly aliased, with the expected output:

```
alias rm='rm -i'
  /bin/rm
```

If you did the optional step and everything seems okay when logging in through another window, you can logout from both windows, and login again to bluenose, and go to the `lab4` directory in which we started.

Step 5: Writing some simple C programs. Now, let us write a program in C. Use `emacs` to edit the C program that prints "hello, world" on a separate line. Use `hello.c` as its filename. Make sure to include the return statement in the main function. For example, the program could be as follows:

```
/* hello.c */
#include <stdio.h>

int main() {
  printf("hello, world\n");
  return 0;
}
```

Add the program to SVN.

Copy the program `hello.c` to the files `hello0.c` and `hello1.c`. Modify the program `hello1.c` by replacing the line '`return 0;`' with '`return 1;`'.

Submit the files '`hello.c`', '`hello0.c`', and '`hello1.c`' to the SVN repository.

Step 6: Utility `diff`. The utility `diff` is a very useful tool. It compares two files and prints a list of editing changes that will convert the first file into the second. Later this term, you will use this command to compare the output of your program with output files given in assignment specifications. This utility is also useful for source code management: If you and another person worked on the same source file, and both of you made changes to different parts of this file, you can use `diff` to help you merge your work.

To run this command, simply enter `diff` followed by the pathnames of two files. To understand its output, read pages 93 and 94 on the Unix textbook.

To try this command, perform the following tasks:

6-a) Run
`diff hello.c hello0.c`
There should be no text in stdout, as these two files are identical.

6-b) Run
`diff hello.c hello1.c`
What does the output mean?

6-c) Save the output in the step 6-b) to the file `diff.out` and add this file to SVN. Commit SVN.

Step 7: Compiling C programs. Compile the C programs using following commands:

```
gcc -o hello hello.c
gcc -o hello0 hello0.c
gcc -o hello1 hello1.c
```

In these commands, `gcc` is the compiler, which means "GNU Compiler Collection". Originally named "GNU C Compiler", this has become the standard compiler for many Unix or Unix-like operating systems. The '`-o`' option specifies the executable file name. Thus, after you enter the command, if the file has been successfully compiled, there will be an executable file named `hello`, and also `hello0` and `hello1` in the current working

directory. The compiler automatically sets the permissions for execution, since this is an executable file. Use a Unix command to check the permissions of `hello`. After this, enter
`./hello`
to run it.

Why do we have to give the pathname of this program to run it?

If there are error messages, fix your program.

Try to compile your program without the '`-o`' option, i.e., do not specify '`-o hello`' in your command line. Are you able to find the executable file in your current working directory? It is not the file `hello` since we did not specify the name for the executable file. After you locate this file, execute it.

Add the file `hello` to SVN and commit.

Step 8: Using Emacs to compile. When developing programs, there is often a need to incrementally modify and test them. One way to do it is to modify the program, exit `emacs`, compile it, run it, and then repeat these operations. Again, we would need to open emacs and find the line that we want to modify, and so on. We can try to be more efficient about it. One solution is to open two terminal windows on the same computer, change the current working directories of both to the same directory, and use one for editing and the other for compiling and running the program.

Alternatively, we can learn how to use `emacs` to compile and run a program without opening another terminal. Let us practice this.

We first learn how to compile our program without exiting `emacs`. Following the steps below:

8-a) Open `hello.c` using `emacs`.
8-b) Enter '`M-x compile`' in emacs. Remember that this can be done by pressing 'Esc x compile' or 'Alt-x compile'. If we change our mind, we can press C-g (i.e., Ctrl-g) to abort the function. After 'verb M-x compile,', we press Enter.
8-c) You will be asked to enter the compile command. Use backspace key to delete the default command entirely, and then enter the '`gcc -o hello hello.c`' command given earlier.
8-d) This will split the current `emacs` window into two, and the bottom window will show the message generated by the compiler.
8-e) To go back to the single window mode so that we can see more lines of source code, enter `C-x 1` (Control-x, then 1 (digit one)).
8-f) You can invoke any shell command without exiting `emacs`, and this includes running the compiled code. To do this, enter `M-!` (e.g., `Esc` key followed by the `!` key). Then you can enter any shell command. You can now enter `./hello` and you will see the result in `emacs`.
8-g) Another way to run a program immediately after compilation is to enter `M-x compile` as before, followed by Enter, and then enter the line:
```
gcc -o hello hello.c && ./hello
```
So, we are entering two commands: compilation, and run the program after. We will see later what is the meaning of `&&`. A brief explanation for now is that after running the first command (`gcc`), the second command is run (`./hello`) only if the first one was successful.

One note about `M-x compile`. The shortcut `M-x` can be used to call one of many Emacs functions. The function 'compile' is one of them. You should exit Emacs at this point.

Step 9: Suspending Emacs In the previous question, we learned the `emacs` hot key that allows us to run any command without exiting `emacs`. Now, instead of invoking shell command inside `emacs`, we learn how to send `emacs` to background and then bring it back to the foreground. This is useful if you wish to stop programming to do some other tasks, and then to resume your work later.

To get started, use emacs to open the `hello.c` file. Now, perform the following steps:

9-a) Enter `C-z`. What does this do?
9-b) We are now using the shell. Now, enter commands to compile and run this program.
9-c) Use what you learned in class about job control to bring `emacs` to foreground again. Do NOT simply enter `emacs`! Hint: These two commands will be helpful: `jobs` and `fg`.

Step 10: Examining the Exit Code. You may be interested why there is a 'return' statement in the main function and what happens to the integer used, like in the statement '`return 0;`'. This number is the exit code of the program. Each Unix command returns an exit code to the shell after its execution to indicate whether it has been executed successfully. The general convention is that the exit code 0 is used to indicate a successful execution, and 1 or other non-zero value indicates a failure. This exit code is then stored in the `$?` built-in variable of the shell, which can be displayed using `echo $?`.

Run the three programs again, '`hello`', '`hello0`', and '`hello1`' and check their exit codes. The first two programs should have an exit code 0, and the last one the exit code 1.

You can try this with any Unix command as well. For example, if you run `ls hello.c` the exit code will be 0 because the file exists, but if you try it with a name of a file that does not exist, the return code will be different.

Step 11: Reading about C functions using `man`. The Unix `man` command can be used to display information about C library functions, such as `printf` and `scanf`. However, when you type the command `man printf`, you will see the manual page of a Unix utility called `printf`, not the C function with the same name. Enter the above command, and take a look at the information displayed on the screen:
`man printf`

Why would this happen? This is because the entries in the Unix manual are grouped into sections. Section 1 is for commands and application programs, while Section 3 is for library functions. When we use the `man` command to look for a word, by default, `man` displays the first entry that it finds, starting from Section 1. That is why the manual page for the command `printf` is displayed.

How do we retrieve the entry for the C `printf` function? To do so, we need specify the section number. Press `q` to quit the previous manual page if you have not done so yet. Then, enter the command:
`man 3 printf`
This will bring up the manual page on the C function `printf`. Use space bar and page-up and page-down keys to scroll up and down the entry. You can even see an example. Thus, the `man` command is very useful when we program in C in Unix.

Try this for the C `scanf` function as well.

Step 12: Experimenting with `printf` function. We learned in class about the `printf` function. We have seen the following statement:

```
printf("Profit: %.2f\n", profit);
```

In this example, the first parameter of `printf` is a string literal `"Profit: %.2f\n"`. This string literal is used as the *format string* of the `printf` function in this statement, as it specifies the output format. In this string, `%.2f` is a place holder indicating where the value of `profit` is to be filled in during printing, and in what format this value will be printed. As we learned in class, the official name for this place holder is *conversion specification*. The general form of conversion specification is $\%fm.pls$, where f are optional flags, m is optional minimal width, $.p$ is optional precision, l is optional length modifier, and s is mandatory conversion specifier.

We talked about the rules for conversion specifications in class, and we will experiment with some of them now. Let us start by writing the following program called `testprintf.c`. Use `emacs testprintf.c` and enter the program:

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

```
int main() {
  int value1 = 123, value2 = 12345;

  printf("[%4d]\n", value1);
  printf("[%-4d]\n", value1);
  printf("[%4d]\n", value2);
  printf("[%-4d]\n", value2);

  return 0;
}
```

After entering this file in emacs, save it.

Before trying the program, let us first explain some conversion specifications: `%d` is a conversion specification for decimal integers In `%4d` we use m minimal width, so the integer will be printed with at least 4 characters length; and in `%-4d` we are also using the flag '`-`' for left alignment of an integer. Based on this, we would expect to obtain the following printouts:

```
[ 123]
[123 ]
[12345]
[12345]
```

Compile the program and verify that this is the output.

SVN submission: Add your file `testprinf.c` to SVN. Commit your changes to SVN.

Run SVN commit once more to be sure that all most recent files are submitted to the SVN.

By now, you have finished the required work of this lab.