CS 240: Programming in C
Fall 2020
Homework 0

Due Monday August 31st, 2020 9:00pm

1 Goals

The purpose of this homework is to provide you with a gentle (re-)introduction to the UNIX command line interface.

2 Remote Work

If you are working remotely on this assignment, you will first need to connect to a Purdue CS Linux system. To do so, ensure you have an SSH client installed on your system (see Campuswire if you don't), and then connect to one of the Purdue CS Linux systems. For example, on many systems, you can just run:

```
ssh yourusername@data.cs.purdue.edu
```

at a command prompt or terminal.

3 Shells

Many systems have command line interfaces. On UNIX, one of those interfaces is called the **shell**. There are many different shells available on UNIX systems – KornShell, BASH, sh, zsh, dash, tcsh, etc. For this course, we strongly recommend that you use BASH.

Shells provide a powerful interface to a system, and often include their own support for scripting. That's right, you can write code (scripts) in a shell language too! We won't worry about that for CS 240 too much.

When you open a terminal, or connect remotely to a *NIX system using ssh, you should be presented with a **prompt**. On Purdue CS systems, this prompt usually looks like:

```
data 51 $
```

"data" is the name of the host (computer), followed by the number of commands entered, followed by a "$".

Go ahead and open a terminal on your system now. Note the prompt.

At this prompt, we can issue commands. Often these commands are programs located somewhere on the system (perhaps one that you just wrote and compiled for CS 240!). Sometimes they are "built-in" commands that the shell automatically interprets.

One command is echo. Try running it per below:

```
data 52 $ echo "Hello!"
Hello!
data 53 $
```

We can see that echo simply prints whatever we tell it to the terminal, and exits, returning us to the command prompt.

4 Setting Up Your Environment

If you would like to set up an environment using BASH automatically, we have provided a script for you. Running this script will overwrite your .bashrc and .vimrc files, and prompt you so that you can set your shell to /bin/bash. If you have already customized your shell environment, you probably do **not** want to run the init command. To setup a fresh environment for cs240, execute the following command:

```
data 54 $ ~cs240/bin/init240
Welcome to CS 240!
Changing login shell for turkstra on NIS master lore.cs.purdue.edu.
New shell:
```

At this point, you must type "/bin/bash" without the quotes and press enter.

```
Changed login shell on NIS master lore.cs.purdue.edu
Your environment is now configured.
Please log out and back in (or close and re-open your terminal).
$
```

5 The File Systems

Understanding how *NIX systems work requires an understanding of what a file system is and how it is structured. A file system dictates how data is organized on a computer system. Everything in *NIX is either a file or a process.

File systems are hierarchical – directories, which are a special type of file, store lists of files and other directories. Paths specify how to traverse this hierarchy to reach a certain point.

When running a shell, you are always located somewhere in this hierarchy. You can find out where with the pwd command – present working directory.

```
data 54 $ pwd
/homes/turkstra
data 55 $
```

In UNIX, every path starts at the root directory — /. So, we can see that the above path starts at the root, goes to the "homes" directory, and then the "turkstra" directory.

There are a lot of files and directories in a typical UNIX environment. Here's a fraction of that hierarchy:

```
/
├── bin
│   └── echo
├── boot
│   ├── initramfs-4.19.12-301.fc29.x86_64.img
│   └── vmlinuz-4.19.12-301.fc29.x86_64
├── dev
│   ├── console
│   ├── random
│   ├── sda1
│   ├── tty0
│   └── urandom
├── etc
│   ├── bashrc
│   ├── inittab
│   ├── rc.d
│   ├── shadow
│   ├── shells
│   ├── sysconfig
│   └── vimrc
├── homes
│   ├── bxd
│   ├── cs240
│   │   └── public
│   │       └── style_sample.c
│   ├── gba
│   ├── grr
│   └── turkstra
│       └── something
│           └── here
├── lib -> usr/lib
├── lib64 -> usr/lib64
├── mnt
├── proc
│   ├── cpuinfo
│   ├── crypto
│   └── zoneinfo
├── root
├── sbin -> usr/sbin
├── sys
│   ├── kernel
│   └── power
├── tmp
│   └── tmp-i3m.xpi
├── usr
│   ├── bin
│   ├── include
│   ├── lib64
```

```
│   ├── local
│   ├── sbin
│   ├── share
│   └── tmp -> ../var/tmp
└── var
    ├── cache
    ├── lib
    ├── local
    ├── mail -> spool/mail
    └── yp
```

You'll learn more about this hierarchy in CS 252. For now, we are mostly concerned with things in /homes. Specifically, your own home directory (per above), and the cs240 home directory, which will contain the files that you need to complete homework assignments.

There are two ways to refer to paths in the file system. The first way is absolute, and always begins with the root directory. So, for example, we could refer to turkstra's home directory like so:

```
/homes/turkstra
```

This is an absolute path – it starts at the root, and only contains the names of other directories and/or files.

The second way to refer to a path is "relative." Paths, for example, can be relative to the current directory. So, for example, if our pwd was /homes/turkstra, and we specified the path:

```
something/here
```

That path would be relative to the pwd and correspond to the following absolute path:

```
/homes/turkstra/something/here
```

There are a few "special" directories that we can use when creating relative paths. The first is "." A single period refers to the current directory, so instead of something/here, we could also say:

```
./something/here
```

Another is ".." which refers to the parent directory. So, for example, given the same pwd as before,

```
../something/here
```

would correspond to:

```
/homes/something/here
```

You can do fun things like this, too:

```
./something/../something/../something/here
```

That path also corresponds to:

```
/homes/turkstra/something/here
```

:-)

Finally, we have "~". ~ by itself means "your home directory". So,

`~/something/here`

Once again corresponds to:

`/homes/turkstra/something/here`

You can also specify a username after the `~`. So, for example, `~cs240`. This would then correspond to:

`/homes/cs240`

Neat!

`~cs240/something/there`

would correspond to:

`/homes/cs240/something/there`

Note that this is different from `~/cs240. ~/cs240` would expand to:

`/homes/turkstra/cs240`

This is an important difference!

Sometimes one might wish to know where a particular command is located in the file system. For this, there is the "which" command.

```
$ which echo
/bin/echo
$
```

This tells us that echo is located in the `/bin` directory.

6 Basic Commands

So, we can make all of these paths, what can we do with them? There are a number of basic commands that are important to know when navigating a UNIX system. Commands like: man, cd, pwd, ls, mkdir, cp, mv, and rm.

The first command in that list, man, is exceedingly useful. "man" is short for "manual," and can be used to find information about almost all commands and programs on a UNIX system.

For example, we can run:

`$ man man`

To learn more about the man command! Don't be too intimidated by how dense some of the material is. man usually displays a brief description of the command followed by the various ways it can be invoked (and arguments), followed by a more detailed description. It may also provide examples.

We already discussed pwd above. What if we want to change directories? Well, that is what **cd** is for. cd can take a relative or an absolute path. If no argument is specified, cd by itself is equivalent to "cd ~"

We can also use the **mkdir** command to create directories. Let's create a cs240 subdirectory inside of your home directory.

```
$ cd
$ mkdir cs240
$ pwd
/homes/turkstra
$ cd cs240
$ pwd
/homes/turkstra/cs240
```

We will work inside of this directory for the rest of the semester.

Now, use the mkdir command to create a "hw0" subdirectory inside your cs240 subdirectory. Change into that directory once you have created it.

One fun command is **touch**. touch can be used to create an empty file, or update the timestamp on a file to be the current date/time. Run the following command:

```
$ touch my_file
```

We have created a number of directories and a file now. It would be nice if we could actually see them. Unsurprisingly, there is a command for that – **ls**.

Take a look at its man page:

```
$ man ls
```

Now, let's list the contents of our pwd:

```
$ ls
my_file
$
```

We see a single file in our directory. Let's make another one:

```
$ touch another_file
$ ls
another_file my_file
$
```

Now there are two! ls supports a number of arguments (you can see them in the man page). One of them is "-l":

```
$ ls -l
total 0
-rw------- 1 turkstra turkstra 0 Jan  7 14:01 another_file
-rw------- 1 turkstra turkstra 0 Jan  7 14:00 my_file
$
```

This the "long listing" format. It shows file permissions, link count, ownership and group information, file size, date and time, and the file name. Let's write something to one of the files:

```
$ echo ‘‘Hello’’ > my_file
```

Note that instead of typing "my_file" (or even "echo"), you can use something called **tab completion**. Type, for example, "m", and then press tab! If there are multiple matches, you can press tab twice to see them.

Placing the > character after a command redirects that command's output to the specified file. It overwrites anything that may already exist, so be careful!

Anyway, let's look at the files again:

```
$ ls -l
total 4
-rw------- 1 turkstra turkstra 0 Jan  7 14:01 another_file
-rw------- 1 turkstra turkstra 6 Jan  7 14:05 my_file
$
```

Notice that instead of a 0 before the date for my_file, we now have a 6! The file now contains 6 bytes. 5 for "Hello", and one extra to indicate new line!

We could use the **cp** command to create a copy of this file too:

```
$ cp my_file my_file2
$ ls -l
total 8
-rw------- 1 turkstra turkstra 0 Jan  7 14:01 another_file
-rw------- 1 turkstra turkstra 6 Jan  7 14:05 my_file
-rw------- 1 turkstra turkstra 6 Jan  7 14:54 my_file2
$
```

Incidentally, there is a command, cat, that can be used to view a file's contents:

```
$ cat my_file2
Hello
$
```

Check out its man page!

Another useful feature of shells is that they retain a certain amount of history about the commands that were most recently run. This history can be navigated using the arrow keys. Imagine, for example, that you'd just finished typing a really complex command, only to realize after pressing enter that you'd given the wrong file name for a parameter. You can press the up arrow key to retrieve the command you just ran and correct only the one mistake, rather than having to re-type the whole thing.

Try pressing the up arrow key now. You should see `$ cat my_file2` appear.

The arrow keys are useful for traversing recent history, but what if there was a command you performed further into the past? Pressing the up arrow key dozens of times may not be any faster than just re-typing it. Fortunately, BASH comes with a way of searching your command history. By pressing `ctrl+R`, the command prompt turns into a search engine that displays the most recent command starting with letters matching those that you've typed. For example, if you type `tou` after entering the search mode (pressing `ctrl+R`), you should see `touch another_file` appear to the right, as that was the most recent command starting with the string `tou`. However, if you continue typing `touch m`, you should now see that it auto-completes to `touch my_file` instead. Once the desired command is displayed on the right, you can press enter to run the command immediately.

Alternatively, pressing escape will paste the command into the command prompt, allowing you to make changes before running it.

Moving on from that brief diversion into command history, let's add another line to our second copy:

```
$ echo "Another line!" >> my_file2
```

Notice this time we use two >'s. Two mean append to the file, instead of overwriting (if you use just one >).

```
$ cat my_file2
Hello
Another line!
$
```

Try running ls to see the other files.

It is possible to move a file too, using **mv**. This can also be used to rename a file. Let's suppose that we accidentally create a file in our home directory, instead of where we want it to be – inside cs240/hw0:

```
$ cd ~
$ touch oops_file
```

After noticing our mistake, we could do something like:

```
$ mv oops_file ~/cs240/hw0
```

Or, if our pwd is already ~, which it should be:

```
$ v oops_file cs240/hw0
```

Don't forget about tab completion! Instead of typing out oops_file, you can probably type "o", or maybe "oo" and hit **tab**.

Anyway, cd back into your hw0 subdirectory:

```
$ cd cs240/hw0
```

Run ls and note that oops_file now resides here.

Finally, lets say that we don't need oops_file anymore. We can delete it:

```
$ rm oops_file
```

Now if you run ls, you will see that the file is gone.

We can remove directories too:

```
$ mkdir oops_dir
$ rmdir oops_dir
```

They must be empty though:

```
$ mkdir oops_dir
$ touch oops_dir/haha
$ rmdir oops_dir
```

```
rmdir: failed to remove 'oops_dir': Directory not empty
$
```

You can recursively remove a directory, but be careful!

```
$ rm -r oops_dir
$
```

7 Graded Questions

Inside of the cs240 home directory – **not** the cs240 subdirectory in your home directory – there is a subdirectory named "public" inside of that subdirectory is a file named "style_sample.c".

Use your newfound knowledge to copy that file into your hw0 directory.

pwd, cd, ls, and cp may all be of use to figure out where things are and copy them to the right location.

Create a file named "answers" and place answers to the following questions in it, along with the output of pwd. You may wish to review the vi(m) refresher at the end of this document:

Question 1: Write the output of `pwd`.

Question 2: Write the copy command that you used.

In addition, there is also a file named "style_sample.h" in the aforementioned "public" directory. Copy that into your hw0 directory as well.

Question 3: Now, diagram the entire file system hierarchy in ~/cs240. Include **all** subdirectories and files. Hint: use ls and cd a lot. "cd .." might be particularly useful...

Question 4: Next, describe, in your own words the difference between an absolute and relative path.

8 Compiling Programs

In this course, we will exclusively use gcc—GNU C compiler. We will use gcc with at least three arguments: -Wall, -Werror, and -std=c99.

If we look at the man page for gcc, we can see that -Wall has the following description: "This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros."

Next, if we take a look at the -Werror argument, we can see this description: "Make all warnings into errors."

Well-written C code, aside from certain specific situations, should not generate any warnings when compiling. These two arguments ensure us that if any warning is encountered, compilation is halted. We will use these arguments when grading your programs. You should always be sure to use them in this course.

Finally, -std is also pretty straightforward: "Determine the language standard."

We will use the C99 standard for this course.

Great! Let's compile a program!

Change into your hw0 subdirectory (if you are not already there), where a copy of the style_sample.c file should reside per above.

Run the following to compile the program:

```
$ gcc -Wall -Werror -std=c99 style_sample.c -lm
```

We need the final argument, -lm, because the program uses the math library. -lm means link against libm (math).

Unfortunately, you can see this did not work...

```
style_sample.c: In function ‘main’:
style_sample.c:65:5: error: implicit declaration of function ‘exit’
[-Werror=implicit-function-declaration]
     exit(ERROR);
     ^~~~
style_sample.c:65:5: error: incompatible implicit declaration of built-in function
‘exit’ [-Werror]
style_sample.c:65:5: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
style_sample.c:79:5: error: incompatible implicit declaration of built-in function
‘exit’ [-Werror]
     exit(ERROR);
     ^~~~
style_sample.c:79:5: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
style_sample.c:86:5: error: incompatible implicit declaration of built-in function
‘exit’ [-Werror]
     exit(ERROR);
     ^~~~
style_sample.c:86:5: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
cc1: all warnings being treated as errors
```

Sometimes compiler errors can be a little cryptic and difficult to decipher. Always start with the first error. It is possible that later errors are a direct result of an earlier error, and will disappear once the first one is fixed. This is not always the case, but the only way to know for sure is to address the errors in the order that they occur.

Thankfully in this case the error is fairly straightforward. It even tells us what to do...

style_sample.c:65:5: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’

Go ahead and edit style_sample.c using your preferred editor. We recommend **vi** or **vim**. Hopefully you have some experience with an editor from CS 180. If you need help remembering, we've included a brief Vi(m) refresher below.

Underneath line 9, which is "`#include <stdio.h>`", add a new line:

```
#include <stdlib.h>
```

Save the file, and attempt to compile it again:

```
$ gcc -Wall -Werror -std=c99 style_sample.c -lm
$
```

This time it should work!

Because we did not specify an output file, gcc defaults to "a.out". We can execute the newly generated binary like so:

```
$ ./a.out
Welcome to the exp (x) Taylor series calculator
***********************************************

Please enter a value x
followed by a Return or and Enter
x:
```

Go ahead and fiddle around with the program.

The ./ is required because by default, the pwd is not part of the shell's PATH variable. The PATH variable dictates where the shell will search for commands. You can see it be executing:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/
games
```

If you'd like, you can add the pwd to the PATH variable so you no longer have to precede commands in the pwd with "./".

This is done by editing ~/.bashrc and adding the following line at the bottom:

```
export PATH=$PATH:.
```

Doing the above is optional.

Now, create a file named "hello.c" and write a C program that displays "Hello, World!" when compiled and executed. Your book may be useful at this point.

If done right, we should be able to compile it like so:

```
$ gcc -Wall -Werror -std=c99 -o hello hello.c
```

Notice that we added a new flag - "-o". This tells the compiler to name the generated binary after the specified argument, instead of naming it "a.out". Now, you can run your program:

```
$ ./hello
Hello, World!
$
```

9 Code Standard Linter

As you should know by this point, we have a code standard that must be followed in CS 240. The file that you copied earlier – style_sample.c – is an example program that follows this code standard. Spend some time looking it over. It is a good reference moving forward.

Part – and **only part** – of the code standard can be checked using a linter that we provide. That linter resides in the bin subdirectory in the cs240 home directory. We can run it like so:

```
$ ~cs240/bin/linter style_sample.c
```

And it should hopefully pass all available tests.

Remember, just because you pass the linter, does **not** mean you have followed the entire code standard. Parts of it are assessed by graders. It is your job to learn the standard and follow it.

As part of the init script, we added ~cs240/bin to your PATH. So, you can also run the linter like so:

```
$ linter style_sample.c
```

Isn't that easier?

10 Submitting Homework

Typically for homework assignments we will have you clone a private git repository into your home directory. We do not expect you to know anything about git for this course.

Let's set up something for today's lab. First, lets move your existing hw0 directory somewhere else:

```
$ cd ~/cs240
$ mv hw0 hw0-old
```

Next, we will clone the repository into your cs240 directory:

```
$ pwd
/homes/turkstra/cs240
$ git clone ~cs240/repos/$USER/hw0
```

$USER is another shell variable. It contains your username...

```
$ echo $USER
turkstra
$
```

This creates a hw0 subdirectory inside the pwd (your cs240 directory). If we provide any skeleton files, those will be included.

For this assignment, we just provide a Makefile:

```
$ cd hw0
$ ls
Makefile
$
```

Copy the files "hello.c" and "answers" from the hw0-old subdirectory into your newly cloned hw0 subdirectory. We'll let you figure out the command. Ask a TA if you need help.

Write the command that you used below, along with that output of pwd:

Question 6: Write the output of `pwd`.

Question 7: Write the copy command that you used.

From here, submitting the homework is simple. Simply run:

```
$ make submit
```

You can confirm your submission by running:

```
$ make verify
```

That's it! We hope that you have enjoyed this episode of CS 240: The Awesome.

Please note that this is a cursory exploration of the rabbit hole that is the *NIX command line interface. It goes much, much deeper and leads to a wonderful land where you can accomplish virtually anything you want – often things that are impossible in the illusory world of graphical user interfaces. :-)

11 vi(m) Refresher

Vim is Vi Improved, a programmer's text editor.

It has a somewhat steep learning curve, but as you learn the keyboard commands, the speed at which one can manipulate text increases significantly. This is because everything in vim is a command or a concatenation of commands. Learning a new command can increase the number of ways you can edit your text exponentially!

What follows is a brief description of commonly used commands. We encourage you to try out each one on your own.

To edit a file, simply execute:

```
$ vi some_file
```

The first thing that you would probably like to know is how to get out.

To do this, we'll have to issue a command. To issue a command, you must be in command mode. To put vi in command mode, press ':' (without quotes).

There are a number of commands that will cause vi to exit:

:q! will quit immediately **without saving anything**.

:q will only quit if there is no unsaved work

While working on things, you may wish to periodically save them without quitting. This is done with the command

:w

As mentioned earlier, vim is all about concatenation of commands. So, we can combine this with the q command to exit and save your work:

:wq

To save a keystroke, you can also use

:x

If you find yourself in command mode, and want to get out (after pressing ':'), simply press escape.

Great, we can quit and save. How do we actually input text, though? Well, to do that you must put vi into insert mode. This is done by pressing a lower case 'i' (again, no quotes).

In insert mode, one can simply type text as usual. To get out, just like when in command mode, you can press the escape key. There are other ways to enter insert mode as well. Pressing 'R' allows you to overwrite text, for example. Again, escape gets you out.

One often used command is to press 'o'. This inserts a new line below the current one, and automatically enters insert mode.

To navigate around, you can use the arrow keys, or when not in insert or command mode, you can use h to move left, l to move right, k to move up, and j to move down.

One of the motivations to use the letter keys instead of arrow keys is that you can avoid moving your fingers off of the home keys. Vi(m) has been created with productivity in mind, so it tries to minimize excessive hand movements. It also does not require the use of a mouse

You can also use ctrl-f to move a page down and ctrl-u to move up a page. (PgUp and PgDn keys also work!)

Again, we can use the power of concatenation and use a number followed by 'j' to move down that many lines - '5j' for example to move down 5 lines.

If you have a specific line number that you would like to go to (maybe obtained from gdb), enter that number followed by 'G'.

5G for example jumps to line 5

If you would like to move the cursor to the beginning of the current line, press '0'. To move to the end, press '$'.

You can move forward and backward entire words by using 'w' and 'b'.

To delete a line of text, you can press 'dd'. To delete the next word, 'dw'. To delete the previous word, 'db'. To delete from the current position to the end of the line, 'd$'.

To replace a word, 'cw' is often useful.

vi supports copying (yanking) and pasting as well.

'yy' yanks the current line. 'y$' yanks to the end of the line. To yank, say, ten lines, you would type '10yy'.

Once you have yanked something, you can paste it by pressing 'p' to go below the cursor, or 'P' to go above.

What if you don't know how many lines to yank? Well, you can mark lines!

To set a mark, you type 'm' followed by some letter. For example, 'mm'. This creates a mark named 'm'.

Then you can scroll some place below that mark, and yank up to that mark by using "y'm". Ignore the double quotes. This means yank up to mark m.

After you've yanked the text, you can paste it as usual.

Too complicated? Vim has a "visual mode" that you can enter by pressing 'v'. All you have to do then is highlight the text of interest and yank it.

Have you made a mistake? 'u' will undo it. Want to redo what you just undid? Press ctrl-r.

Want to jump to the end of a file? 'G' is your friend!

Back to the beginning? 'gg'

These are the basics. There are **many** other commands.

Look around on the web – you'll find lots of great resources! And, don't hesitate to ask us questions!