HW 0: **Introduction to CS 162**

CS 162

Due: September 6, 2019

Contents

This semester, you will be using various tools in order to submit, build, and debug your code. This assignment is designed to help you get up to speed on some of these tools.
This assignment is due at 9:00 pm on 9/6/2019.

1 Setup

1.1 GitHub and the Autograder

Code submission for all projects and homework in the class will be handled via GitHub so you will need a GitHub account. We will provide you with private repositories for all your projects. **You must not use your own public repositories for storing your code. Throughout the course, if you discover repositories with CS 162 solutions, please notify the course staff. Using solutions you may discover on-line is not permitted. Seek course staff for help. What you turn in should reflect your work.** Visit cs162.eecs.berkeley.edu/autograder[1] to register your GitHub account with the autograder.

1.2 Vagrant

We have prepared a Vagrant virtual machine image that is pre-configured with all the tools necessary to run and test your code for this class. Vagrant is an tool for managing virtual machines. You can use Vagrant to download and run the virtual machine image we have prepared for this course. (The virtual machine for the course is new this term. Do not use one from a previous semester.)

Note: If you do not want to set up Vagrant on your own machine, take a look at the CS162 VM provisioner[2] on GitHub for more options. You can run the VM on a variety of hypervisors, cloud computing platforms, or even on bare metal hardware.

(If you are using Windows, these steps may or may not work. If they do work, you should be fine. If they don't work—which is likely to happen if you have an older version of Windows that doesn't support SSH on the command line—then skip to the section below labeled "Windows").

1. Vagrant depends on VirtualBox (an open source virtualization product) so first you will need to download and install the latest version from the VirtualBox website[3]. **We have observed that certain earlier versions of VirtualBox, specifically versions 6.0.0 to 6.0.4, do not properly boot our class VM. We recommend using version 6.0.10, the latest version of VirtualBox at the time of writing.** We will talk in class about virtual machines, but you can think of it as a software version of actual hardware.

2. Now install the latest version of Vagrant from the Vagrant website[4].

3. Once Vagrant is installed, type the following into your terminal:

```
$ mkdir cs162-vm
$ cd cs162-vm
$ vagrant init cs162/fall2019
$ vagrant up
$ vagrant ssh
```

These commands will download our virtual machine image from our server and start a ssh session. The "up" command will take a while, and may require an Internet connection.

[1]https://cs162.eecs.berkeley.edu/autograder
[2]https://github.com/Berkeley-CS162/vagrant/
[3]https://www.virtualbox.org/wiki/Downloads
[4]http://www.vagrantup.com/downloads.html

4. You need to run all vagrant commands from the cs162-vm directory you created earlier. Do NOT delete that directory, or vagrant will not know how to manage the VM you created.

5. You can run `vagrant halt` to stop the virtual machine. If this command does not work, make sure you are running it from your host machine, not inside SSH. To start the virtual machine the next time, you only need to run `vagrant up` and `vagrant ssh`. All of the other steps do not need to be repeated.

1.2.1 Windows (OS X and Linux users can skip this section)

Your Windows installation may not support SSH from the command line, especially if you do not have the latest version. In this case, the "vagrant ssh" command from the above steps will cause an error message prompting you to download Cygwin or something similar that supports an ssh client. Here[5] is a good guide on setting up Vagrant with Cygwin in windows.

Alternatively, it is possible to use PuTTY instead of Cygwin, but this might be slightly more work to set up.

If you get an error about your VM bootup timing out, you may need to enable VT-x (virtualization) on your CPU in BIOS.

1.2.2 Troubleshooting Vagrant

If "`vagrant up`" fails, try running "`vagrant provision`" and see if it fixes things. As a last resort, you can run "`vagrant destroy`" to destroy the VM. Then, start over with "`vagrant up`".

1.2.3 Git Name and Email

Run these commands to set up your Name and Email that will be used for your Git commits. Make sure to replace "Your Name" and "your_email@berkeley.edu" with your REAL name and REAL email.

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your_email@berkeley.edu"
```

1.2.4 ssh-keys

You will need to setup your ssh keys in order to authenticate with GitHub from your VM.

New GitHub Users

SSH into your VM and run the following:

```
$ ssh-keygen -N "" -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub
```

The first command created a new SSH keypair. The second command displayed the public key on your screen. You should log in to GitHub and go to github.com/settings/ssh[6] to add this SSH public key to your GitHub account. The title of your SSH keypair can be "CS 162 VM". The key should start with "ssh-rsa" and end with "vagrant@development".

[5] https://gist.github.com/rogerhub/456ae31427aafe5b70f7
[6] https://github.com/settings/ssh

Experienced GitHub Users

If you already have a GitHub SSH keypair set up on your local machine, you can use your local ssh-agent to utilize your local credentials within the virtual machine via ssh agent forwarding. Simply use `vagrant ssh` to ssh into your machine. The Vagrant should enable SSH agent forwarding automatically. If this doesn't work, you can also use the instructions in the previous "New GitHub Users" section.

1.2.5 Repos

You will have access to two private repositories in this course: a personal repository for homework, and a group repository for projects. We will publish skeleton code for homeworks in Berkeley-CS162/student0[7] and we will publish skeleton code for group projects in Berkeley-CS162/group0[8]. These two skeleton code repositories are already checked out in the home folder of your VM, inside `~/code/personal` and `~/code/group`.

You will use the "Remotes" feature of Git to pull code from our skeleton repos (when we release new skeleton code) and push code to your personal and group repos (when you submit code). Your working files will be stored within the VM. Back them up by pushing to your github repo. Save your work early and often. Several small clear commits and pushes is good practice. Communication with course staff will often involve looking at the code and commits in your repo.

The Git Remotes feature allows you to link GitHub repositories to your local Git repository. We have already set up a remote called "staff" that points to our skeleton code repos on GitHub, for both your personal and group repo. You will now add your own remote that points to your private repo so you can submit code.

You should have received the link to your personal private GitHub repo when you registered with the autograder earlier. Add a new remote by doing the following steps in your VM:

1. First cd into your personal repository

   ```
   cd ~/code/personal
   ```

2. Then visit your personal repo on GitHub and find the SSH clone URL. It should have the form "git@github.com:Berkeley-CS162/..."

3. Now add the remote

   ```
   git remote add personal YOUR_GITHUB_CLONE_URL
   ```

4. You can get information about the remote you just added

   ```
   git remote -v
   git remote show personal
   ```

5. Pull the skeleton, make a test commit and push to `personal master`

   ```
   git pull staff master
   touch test_file
   git add test_file
   git commit -m "Added a test file."
   git push personal master
   ```

[7] https://github.com/Berkeley-CS162/student0/
[8] https://github.com/Berkeley-CS162/group0/

In this course, “master” is the default Git branch that you will use to push code to the autograder. You can create and use other branches, but only the code on your master branch will be graded. You should do this test commit before Monday. We want to know that everyone has got this basic infrastructure in place.

6. Within 30 minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza). Check cs162.eecs.berkeley.edu/autograder[9] for more information.

1.3 Autograder

Here are some important details about how the autograder works:

- The autograder will automatically grade code that you push to your master branch, UNLESS the assignment you are working on is LATE.
- If your assignment is late, you can still get it graded, but you will be using slip days. You can request late grading using the autograder’s web interface at cs162.eecs.berkeley.edu/autograder[10].
- Your final score will be the best score that you received in any build. So, even if you push code that fails more autograder tests than before, it will not decrease your score. (But, still you should use the autograder sparingly. See below.)
- If your score does not improve when you push new code, it will not affect your slip days. So, if you request a build after the deadline that does not improve your score, you will not have to use slip days.

The autograder is for grading, not for testing. You should develop and carry out your tests in your local environment. Lots of spurious autograder submissions can interfere with people getting their work done. And the turnaround time is too slow for testing. It provides final confirmation that your tests are consistent with ours.

1.4 Editing code in your VM

The VM contains a SMB server that lets you edit files in the vagrant user’s home directory. With the SMB server, you can edit code using text editors on your host and run git commands from inside the VM. **This is the recommended way of working on code for this course**, but you are free to do whatever suits you best. One possibility is just using a non-graphical text editor in an SSH session.

1.4.1 Windows

1. Open the file browser, and press `Ctrl L` to focus on the location bar.
2. Type in `\\192.168.162.162\vagrant` and press Enter.
3. The username is **vagrant** and the password is **vagrant**.

You should now be able to see the contents of the vagrant user’s home directory.

[9] https://cs162.eecs.berkeley.edu/autograder
[10] https://cs162.eecs.berkeley.edu/autograder

1.4.2 Mac OS X

1. Open Finder.
2. In the menu bar, select **Go** → **Connect to Server...**.
3. The server address is `smb://192.168.162.162/vagrant`.
4. The username is **vagrant** and the password is **vagrant**.

You should now be able to see the contents of the vagrant user's home directory.

1.4.3 Linux

Use any SMB client to connect to the `/vagrant` share on 192.168.162.162 with the username **vagrant** and password **vagrant**. Your distribution's file browser probably has support for SMB out of the box, so look online for instructions about how to use it.

1.5 Shared Folders

The `/vagrant` directory inside the virtual machine is connected to the home folder of your host machine. You can use this connection if you wish, but the SMB method in the previous section is recommended. (You can also learn more about the file system of your local machine by finding where the file system of your VM is mounted. Can you find it?)

2 Useful Tools

Before continuing, we will take a brief break to introduce you to some useful tools that make a good fit in any system hacker's toolbox. Some of these (git, make) are MANDATORY to understand in that you won't be able to compile/submit your code without understanding how to use them. Others such as gdb or tmux are productivity boosters; one helps you find bugs and the other helps you multitask more effectively. All of these come pre-installed on the provided virtual machine. They are ESSENTIAL.

Note: We do not go into much depth on how to use any of these tools in this document. Instead, we provide you links to resources where you can read about them. We highly encourage this reading even though not all of it is necessary for this assignment. We guarantee you that each of these will come in handy throughout the semester. If you need any additional help, feel free to ask any of the TA's at office hours!

2.1 Git

Git is a version control program that helps keep track of your code. GitHub is only one of the many services that provide a place to host your code. You can use git on your own computer, without GitHub, but pushing your code to GitHub lets you easily share it and collaborate with others.

At this point, you have already used the basic features of git, when you set up your repos. But an understanding the inner workings of git will help you in this course, especially when collaborating with your teammates on group projects.

If you have never used git or want a fresh start, we recommend you start here[11]. If you sort of understand git, this presentation[12] we made and this website[13] will be useful in understanding the inner workings a bit more.

2.2 make

make is a utility that automatically builds executable programs and libraries from source code by reading files called Makefiles, which specify how to derive the target program. How it does this is pretty cool: you list dependencies in your Makefile and make simply traverses the dependency graph to build everything. Unfortunately, make has very awkward syntax that is, at times, very confusing if you are not properly equipped to understand what is actually going on.

A few good tutorials are here[14] and here[15]. And of course the official GNU documentation (though it may be a bit dense) here[16].

For now we will use the simplest form of make: without a `Makefile`. (But you will want to learn how to build decent Makefiles before long!) You can compile and link `wc.c` by simply running:

```
$ make wc
```

This created an executable, which you can run. Try

```
$ ./wc wc.c
```

How is this different from the following? (Hint: run "`which wc`".)

```
$ wc wc.c
```

[11]http://git-scm.com/book/en/Getting-Started
[12]http://goo.gl/cLBs3D
[13]http://think-like-a-git.net/
[14]http://wiki.wlug.org.nz/MakefileHowto
[15]http://mrbook.org/blog/?s=make
[16]http://www.gnu.org/software/make/manual/make.html

2.3 man

`man` - the user manual pages - is really important. There is lots of stuff on the web, but `man` is definitive. The warm up to your first assignment is going to be to modify `wc.c`, so that it implements word count according to the specification of "`man wc`", except that it does not need to support any flags and only needs to support a single input file, (or `STDIN` if none is specified). Beware that `wc` in OS X behaves **differently** from `wc` in Ubuntu. We will expect you to follow the behavior of `wc` in Ubuntu, i.e., in your VM.

2.4 gdb

Debugging C programs is hard. Crashes don't give you nice exception messages or stack traces by default. Fortunately, there's gdb. If you compile your programs with a special flag `-g` then the output executable will have debug symbols, which allow gdb to do its magic. If your run your C program inside gdb it will allow you to not only look get a stack trace, but inspect variables, change variables, pause code and much more!

Normal gdb has a very plain interface. So, we have installed `cgdb` for you to use on the virtual machine, which has syntax highlighting and few other nice features. In cgdb, you can use `i` and `ESC` to switch between the upper and lower panes.

gdb can start new processes and attach to existing processes (which will be useful when debugging your project.)

This[17] is an excellent read on understanding how to use gdb.

Again, the official documentation[18] is also good, but a bit verbose.

Take a moment to begin working on your `wc`. Provide the -g flag when you compile your program with `gcc`. Start the program under gdb. Set a break point at main. Run to there. Try out various commands. Figure out how to pass command line arguments. Add local variables and try probing their values. Learn about step, next, and break.

2.5 tmux

tmux is a terminal multiplexer. It basically simulates having multiple terminal tabs, but displays them in one terminal session. It saves having to have multiple tabs of sshing into your virtual machine.

You can start a new session with `tmux new -s <session_name>`

Once you create a new session, you will just see a regular terminal. Pressing `ctrl-b + c` will create a new window. `ctrl-b + n` will jump to the nth window.

`ctrl-b + d` will "detach" you from your tmux session. Your session is still running, and so are any programs that you were running inside it. You can resume your session using `tmux attach -t <session_name>`. The best part is this works even if you quit your original ssh session, and connect using a new one.

Here[19] is a good tmux tutorial to help you get started.

2.6 vim

vim is a nice text editor to use in the terminal. It's well worth learning. Here[20] is a good series to get better at vim. Others may prefer emacs. Whichever editor you choose, you will need to get proficient with an editor that is well suited for writing code.

If you want to use Sublime Text, Atom, CLion, or another GUI text editor, look at 1.4 Editing code in your VM, which shows you how to access your VM's filesystem from your host.

[17] http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html
[18] https://sourceware.org/gdb/current/onlinedocs/gdb/
[19] http://danielmiessler.com/study/tmux/
[20] http://derekwyatt.org/vim/tutorials/

2.7 ctags

ctags is a tool that makes it easy for you to navigate large code bases. Since you will be reading a lot of code, using this tool will save you a lot of time. Among other things, this tool will allow you to jump to any symbol declaration. This feature together with your text editor's go-back-to-last-location feature is very powerful.

Instructions for installing ctags can be found for vim here[21] and for sublime here[22]. If you don't use vim or sublime, ctags still is probably supported on your text editor although you might need to search installation instructions yourself.

3 Warm Up

Before an assignment, we will often have a set of *warmups*. Feel free to take a quick look at the assignment first. If the assignment looks daunting then we would strongly recommend you try the warm up first.

The processing for submitting warm ups is the same as the process for submitting your regular assignments and we will track who attempts or completes the warm ups, but they are ultimately optional.

We will often also provide *stretches* that allow you to go a bit further, if you are finding the assignment not too challenging.

3.1 wc

We are going to use `wc.c` to get you thinking once again in C, with an eye to how applications utilize the operating system - passing command line arguments from the shell, reading files, and standard file descriptors. All of these things you encountered in CS61C, but they will take on new meaning in CS162.

Your first task to write a clone of the tool `wc`, which counts the number of lines, words, and characters inside a particular text file. You can run the official `wc` in your VM to see what your output should look like, and try to mimic its basic functionality in wc.c.

Just like the real version of `wc` your program should count the number of characters, words, and lines in file(s) provided as input. If no files are provided, it should read from `STDIN`. Additionally, your program should support flags, in particular `-l -w -c` .

While you are working on this take the time to get some experience with gdb. Use it to step through your code and examine variables.

Beware that `wc` in OS X behaves **differently** from `wc` in Ubuntu. We will expect you to follow the behavior of `wc` in Ubuntu.

4 Your First Assignment

4.1 fixit

Being able to read code written by others and spotting the errors will be an important skill both in this class and in the real world.

Your second task is to look at `fixit/uniq.c` and `fixit/sort.c`. They are similar to the UNIX utilities `uniq` and `sort`, respectively. **An important difference is that `uniq.c`, by design, processes the input word-by-word instead of line-by-line.** Each of these files is a standalone program and has a bug in it. For each program, write down the following in `bugs.txt`. Your answers should be short and concise.

1. Briefly describe what the program is intended to do.

[21] http://ricostacruz.com/til/navigate-code-with-ctags.html
[22] https://github.com/SublimeText/CTags

2. Describe the bug(s).
3. How would you reproduce the bug(s)?
4. What are the symptoms of the bug(s)?
5. How did you go about fixing the bug(s)?

There are relatively "clean" solutions to fixing all of these programs, and as a stretch, you should fix them.

4.2 Executables and addresses

Now that you have dusted off your C skills and gained some familiarity with the CS 162 tools, we want you to understand what is really inside of a running program and what the operating system needs to deal with.

4.2.1 gdb

Load up your `wc` executable in gdb with a single input file command line argument, set a breakpoint at `main`, start your program, and continue one line at a time until you are in the middle of your program's execution. Take a look at the stack using `where` or `backtrace` (bt).

While you are looking through gdb, think about the following questions and put your answers in the file `gdb.txt`.

- What is the value of `argv`? (hint: print argv)
- What is pointed to by `argv`? (hint: print argv[0])
- What is the address of the function `main`?
- Try `info stack`. Explain what you see.
- Try `info frame`. Explain what you see.
- Try `info registers`. Which registers are holding aspects of the program that you recognize?

4.2.2 objdump

There is more to the executable than meets the eye. Let's look down inside. Run "`objdump -x -d wc`".

You will see that your program has several segments, names of functions and variables in your program correspond to labels with addresses or values. And the guts of everything is chunks of stuff within segments.

In the objdump output these segments are under the section heading. There's actually a slight nuance between these two terms which you can read more about online.

While you are looking through the objdump try and think about the following questions and put the answers in the file `objdump.txt`.

- What file format is used for this binary? And what architecture is it compiled for?
- What are some of the names of segment/sections you find?
- What segment/section contains `main` (the function) and what is the address of `main`? (It will not be exactly the same as what you saw in gdb. An optional stretch exercise is to think about why. See the answer to @57 on Piazza and the Wikipedia article on relocation[23] for a hint.)
- Do you see the stack segment anywhere? What about the heap? Explain.

[23] https://en.wikipedia.org/wiki/Relocation_(computing)

4.2.3 map

OK, now you are ready to write a program that reveals its own executing structure. The second file in hw0, `map.c` provides a rather complete skeleton. You will need to modify it to get the addresses that you are looking for. The output of the solution looks like the following (the addresses may be different).

```
_main  @ 0x4005c2
recur @ 0x40057d
_main stack: 0x7fffda11f73c
static data: 0x601048
Heap: malloc 1: 0x671010
Heap: malloc 2: 0x671080
recur call 3: stack@ 0x7fffda11f6fc
recur call 2: stack@ 0x7fffda11f6cc
recur call 1: stack@ 0x7fffda11f69c
recur call 0: stack@ 0x7fffda11f66c
```

Now think about the following questions and put the answers in `map.txt`.

- Use objdump with the `-D` flag on the map executable. Which of the addresses from the output of running `./map` are defined in the executable, and which segment/section is each defined in?
- Make a list of the important segments, and what they are used for (look up their names on the Internet if you don't know).
- What direction is the stack growing in?
- How large is the stack frame for each recursive call?
- Where is the heap? What direction is it growing in?
- Are the two `malloc()`ed memory areas contiguous? (e.g. is there any extra space between their addresses?)

4.3 user limits

The operating system needs to deal with the size of the dynamically allocated segments: the stack and heap. How large should these be? Poke around a bit to find out how to get and set these limits on Linux. Modify `limits.c` so that it prints out the maximum stack size, the maximum number of processes, and maximum number of file descriptors. Currently, when you compile and run `limits.c` you will see it print out a bunch of system resource limits (stack size, heap size, ..etc). Unfortunately all the values will be 0. Your job is to get this to print the ACTUAL limits (use the soft limits, not the hard limits). (Hint: run "`man getrlimit`")

You should expect output similar to this:

```
stack size: 8388608
process limit: 2782
max file descriptors: 1024
```

4.4 words

Programming in C is a very important baseline skill for CS 162. This exercise should make sure you're comfortable with the basics of the language. In particular, you need to be fluent in working with `structs`, linked data structures (e.g., lists), pointers, arrays, `typedef` and such, which 61C may have touched only lightly.

You will be writing `words`. `words` is a program that parses files, finds each word that occurs, records a count of how many times each occurs, and prints the words and their counts to `stdout`.

Like most Linux utilities in the real world, your program should read its input from each of the files specified as command line arguments, printing the cumulative word counts. If no file is provided your program should read from `stdin`. Your program should print each unique word as well as the number of times it occured. This should be sorted in order of frequency (low first). The alphabetical ordering of words should be used as a tie breaker.

A word is defined as a sequence of contiguous alphabetical characters of length greater than one. (Our friends 'a' and 'I' will be ignored.) All words should be converted to their lower-case representation and be treated as not case-sensitive.

For example, if we have a file, `words.txt` which contains :

```
abc def AaA
bbb zzz aaa
```

Then `./words words.txt` should print

```
1 abc
1 bbb
1 def
1 zzz
2 aaa
```

Hint: You can run

```
cat <filename>
    | tr " " "\n"
    | tr -s "\n"
    | tr "[:upper:]" "[:lower:]"
    | tr -d -C "[:lower:]\n"
    | sort
    | uniq -c
    | sort -n
```

to verify the basic functionality of your program. But, don't treat this as a testing spec.

In C, `.h` files are how we engineer abstractions. They define the equivalent of objects, types, and methods. The corresponding `.c` file provides the implementation of the abstraction. But you should be able to write code against the `.h` without peeking under the covers at its implementation. (Or rather, peeking is for learning how to build such abstractions, which is really important, but you shouldn't need to rely on it.) All the abstractions that the internet and all its systems are built out of can be found in Linux at `/usr/include/`.

In this case, `word_count.h` provides the signatures of your `word_count` abstraction, including the `struct` that is the object representation and a `typedef` that defines the corresponding type. (You will see both approaches in Pintos.) Your first task is to implement the methods in `word_count.c` and then use them to implement your `words` application.

We have provided you with a compile version of a `sort_words` so that you do not need to write that. It is in the object file `wc_sort.o`. The `Makefile` links this in with your two object files, `words.o` and `word_count.o`. It conforms to `word_count.h` and can be used without having access to the source code, just like all those system libraries that you use.

Note that `words.o` is an ELF formatted binary. As such you will need to use a system which can run ELF executables to test your program (such as the 162 VM). Note that both Windows and OS X do **NOT** use ELF and as such should not be used for testing.

4.5 Stretch

If you are feeling like you'd like to stretch your C practice a bit further, implement `insert_ordered_words` and `sort_words` by using it. This will give you excellent pointer practice.

4.6 Autograder & Submission

To push to autograder do:

```
cd ~/code/personal/hw0
git status
git add gdb.txt objdump.txt map.txt bugs.txt
git add wc/wc.c limits.c map.c words/main.c words/word_count.c Makefile
git commit -m "Finished my first CS 162 assignment"
git push personal master
```

This saves your work and it gives the instructors a chance to see the progress you are making. Congratulations for not waiting until the last minute.

Within a few minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza). Check cs162.eecs.berkeley.edu/autograder[24] for more information.

Your work on `gdb.txt`, `objdump.txt`, `map.txt`, and `bugs.txt` will not be graded by the autograder and instead will be graded manually.

Hopefully after this you are slightly more comfortable with your tools. You will need them for the long road ahead!

[24] https://cs162.eecs.berkeley.edu/autograder