HW 0: **Introduction to CS 162**

CS 162

Due: January 27, 2021

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 11:59 PM PST on 01/27/2021.

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.org/autograder[1] to register your GitHub account with the autograder.

1.2 Virtual Machine

We have prepared a virtual machine image that is preconfigured with all the development tools necessary to run and test your code for this class. Later in the course, we will discuss virtual machines; for now, you can think of it as a software version of actual hardware.

If you are unable to use an x86-based VM (notably Apple M1 Macs), or would prefer not to set up a local development environment, you can use instructional machines instead. See the "Instructional Machine Setup" section for details.

Vagrant is a cross-platform 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 different every term. Do not use one from a previous semester.) Vagrant depends on providers for virtualization. The default provider is VirtualBox, an open-source virtualization product.

First, download and install the latest version of VirtualBox from the VirtualBox website[2]. **We recommend using the latest version of VirtualBox, as certain earlier versions of VirtualBox (6.0.0 - 6.0.4) may not properly boot our class VM. Alternatively, for some users, VirtualBox 5 is more stable than VirtualBox 6.**

Next, download and install the latest version of Vagrant from the Vagrant website[3].

Note: You may choose to use the VM using a different provider or without Vagrant. You can take a look at the CS 162 VM provisioner[4] on GitHub for more options, though these are not officially supported by staff. You can run the VM on a variety of hypervisors, cloud computing platforms, or even on bare metal hardware.

After Vagrant and VirtualBox are installed, type the following into your terminal to initialize your VM and begin an SSH session:

```
$ mkdir cs162-vm
$ cd cs162-vm
$ vagrant init cs162/spring2021
$ vagrant up
$ vagrant ssh
```

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

These commands will download our virtual machine image from our server and boot it. The download of the image will take a while, and requires an Internet connection.

From the `cs162-vm` directory you created earlier, you can run "`vagrant up`" to start the virtual machine, and "`vagrant halt`" to stop the virtual machine. To start a SSH session to the virtual machine, run "`vagrant ssh`". You can run "`exit`" to exit a SSH session.

1.2.1 Vagrant Troubleshooting

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`".

1.2.2 Windows Troubleshooting

Windows Subsystem for Linux (WSL)
We do not recommend using Vagrant and/or VirtualBox in WSL. If you choose to use Vagrant for Linux inside WSL, you should still have VirtualBox installed in Windows. Vagrant has documentation[5] on the configuration necessary to support usage of Windows VirtualBox inside WSL.

SSH Client
On the latest version of Windows 10, the virtual machine should work on the default command line. If you are using an older version of Windows, your installation may not support SSH from the command line. 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. Previous TAs have wrote a guide[6] on setting up Vagrant with Cygwin in Windows. Alternatively, it is possible to use PuTTY instead of Cygwin.

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

Hyper-V
If you get an error about Hyper-V being enabled, you should disable Hyper-V to use VirtualBox. Alternatively, you can set up Vagrant to utilize Hyper-V as the provider instead of VirtualBox, and create the VM yourself (see CS 162 VM provisioner above for details).

1.3 Virtual Machine Configuration

After you connect to your virtual machine with `vagrant ssh`, you should set up Git for your repositories on your virtual machine.

1.3.1 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 name and email.

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

[5] https://www.vagrantup.com/docs/other/wsl
[6] https://gist.github.com/rogerhub/456ae31427aafe5b70f7

1.3.2 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[7] 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".

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. You can enable SSH agent forwarding in your `Vagrantfile` by enabling `config.ssh.forward_agent = true`. Then, you can still use `vagrant ssh` to SSH into your machine. Alternatively, you can use the instructions in the previous "New GitHub Users" section.

1.3.3 Repositories

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[8] and we will publish skeleton code for group projects in Berkeley-CS162/group0[9]. 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. Breaking up the implementation of a large feature into several, small, clear commits is good practice, and will make it easier to determine where in your codebases' history a bug was introduced. 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 that 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, switch 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/..."

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

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
   ```

   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 as soon as possible – we want to know that everyone has 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.org/autograder[10] for more information.

1.4 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.org/autograder[11].
- **Your final score in the autograder is not the maximum of your attempts, but rather your score for only your latest build.** Any non-autograded components, like style and written portions, will be graded based on your last build for the assignment.

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.

1.5 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.

[10] https://cs162.org/autograder
[11] https://cs162.org/autograder

1.5.1 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.

1.5.2 macOS

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.

1.5.3 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.

1.6 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?)

1.7 Instructional Machine Setup (VM Alternative)

If you prefer not to use the VM as a development enironment, or you are unable to set up the VM for local development, you can use the EECS Instructional Computers instead.

Note: This setup is experimental, and primarily applies to students planning to use the new Apple M1 Macs. If you plan on using this setup, please notify course staff (i.e. via private Piazza post) so that we can keep you updated regarding any infrastructure developments.

To use the EECS Instructional Computers, you should first acquire a UNIX account from EECS Instructional WebAcct[12], then use that account to login to a suitable EECS Instructional Computer (ashby, derby, cedar). The first time you login, you should run the CS 162 setup script, which loads the development repositories and utilities. You can then use the instructional machine as your development environment, similar to the VM, though some features may still be unavailable.

```
$ ssh cs162-xxx@ashby.cs.berkeley.edu
$ . ~cs162/public/setup
```

You should still follow the steps in "Virtual Machine Configuration" to set up Git, SSH, and your repositories.

[12] http://inst.eecs.berkeley.edu/webacct

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 of 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[13]. If you sort of understand git, this presentation[14] we made and this website[15] 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[16] and here[17]. And of course the official GNU documentation (though it may be a bit dense) here[18].

2.3 man

`man` – the user manual pages – is really important. There are lots of stuff on the web, but the documentation in `man` is definitive. The `man` pages can be accessed through a terminal. For instance, if you wanted to learn more about the `ls` command, simply type "`man ls`" into your terminal. If you were curious about a function called `fork`, you could learn more about it by typing "`man fork`" into your terminal.

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

2.4 gdb

Debugging C programs is hard. Crashes don't give you nice exception messages or stack traces by default. Fortunately, there's the GNU Debugger, or `gdb` for short. 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 you run your C program inside `gdb`, you will be able to not only look at a stack trace, but also inspect variables, change variables, pause code and much more! Moreover, `gdb` can even start new processes and attach to existing processes (which will be useful when debugging PintOS.)

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.

This[19] is an excellent read on understanding how to use `gdb`. The official documentation[20] is also good, but a bit verbose.

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 `ssh`ing 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[21] is a good `tmux` tutorial to help you get started.

If you use macOS and iTerm2, you can utilize iTerm2's built-in `tmux` integration, which - when you are SSH'ed into your VM - will automatically perform the appropriate tmux commands for you whenever you create a new tab or split using iTerm2's normal tab/split commands (`Cmd+T` and `Cmd+D`). To use this, first enable iTerm2's tmux integration by following these instructions[22], then after SSHing into your VM, add the following lines to the end of your `~/.bashrc`.

```
if [ -n "$SSH_TTY" ] && [ -z "$TMUX" ] && shopt -q login_shell
then
    tmux -CC new -A -s main
fi
```

The next time you SSH into your VM, the tmux integration should be working.

2.6 vim

`vim` is a nice text editor to use in the terminal. It's well worth learning. The `vimtutor` program provides an excellent, beginner-friendly tutorial on how to use vim. You can open it by simply running `vimtutor` inside your VM (note when you first run it, it may display a message along the lines of "The file /usr/share/vim/vim80/tutor/tutor.C... does not exist, creating english version..." – this is okay, just be patient for a few seconds and the tutorial will open). 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 1.5 Editing code in your VM, which shows you how to access your VM's filesystem from your host.

[19] http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html
[20] https://sourceware.org/gdb/current/onlinedocs/gdb/
[21] http://danielmiessler.com/study/tmux/
[22] https://iterm2.com/documentation-tmux-integration.html

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[23] and for sublime here[24]. 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 Your First Assignment

3.1 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 CS 61C may have touched only lightly.

You will be writing a program called `words`. `words` is a program that counts (1) the total amount of words and (2) the frequency of each word in a file(s). It then prints the results to `stdout`. Like most UNIX 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`.

In C, header files (suffixed by `.h`) are how we engineer abstractions. They define the objects, types, methods, and—most importantly—documentation. The corresponding `.c` file provides the implementation of the abstraction. You should be able to write code with the header file without peeking under the covers at its implementation.

In this case, `words/word_count.h` provides the definition of the `word_count` struct, which we will use as a linked list to keep track of a word and its frequency. This has been `typedef`'d into `WordCount`. This means that instead of typing out `struct word_count`, we can use `WordCount` as shorthand. The header file also gives us a list of functions that are defined in `words/word_count.c`. Part of this assignment is to write code for these functions in `words/word_count.c`.

We have provided you with a compiled version of `sort_words` so that you do not need to write the `wordcount_sort` function. However, you may still need to write your own comparator function (i.e. `wordcount_less`). The `Makefile` links this in with your two object files, `words.o` and `word_count.o`.

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 CS 162 VM). Windows and OS X do **NOT** use ELF and as such should not be used for testing.

For this section, you will be making changes to `words/main.c` and `words/word_count.c`. After editing these files, `cd` into the `words` directory and run `make` in the terminal. This will create the `words` executable. (Remember to run `make` after making code changes to generate a fresh executable). Use this executable (and your own test cases) to test your program for correctness. The autograder will use a series of test cases to determine your final score for this section.

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

For the below examples, suppose we have a file called `words.txt` that contains the following content:

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

3.1.1 Total Word Count

Your first task will be to implement total word count. When executed, `words` will print the total number of words counted to `stdout`. At this point, you will not need to make edits to `word_count.c`. A complete implementation of the `num_words()` function can suffice.

A word is defined as a sequence of contiguous alphabetical characters of length greater than one. All words should be converted to their lower-case representation and be treated as not case-sensitive. The maximum length of a word has been defined at the top of `main.c`.
After completing this part, running `./words words.txt` should print:

```
The total number of words is: 6
```

3.1.2 Word Frequency Count

Your second task will be to implement frequency counting. Your program should print each unique word as well as the number of times it occurred. This should be sorted in order of frequency (low first). The alphabetical ordering of words should be used as a tie breaker. The `wordcount_sort` function has been defined for you in `wc_sort.o`. However, you will need to implement the `wordcount_less` function in `main.c`.

You will need to implement the functions in `word_count.c` to support the linked list data structure (i.e. `WordCount` a.k.a. `struct word_count`). The complete implementation of `word_count.c` will prove to be useful when implementing `count_words()` in `main.c`.

After completing this part, running `./words -f 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 (don't treat this as a testing spec though).

3.2 user limits

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.

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: 8000000
process limit: 30000
max file descriptors: 1000
```

You can run `make limits` to compile your code.

3.3 The A-Z's of GDB

Now we're going to use a sample program, `map`, for some GDB practice. The `map` program is designed to print out its own executing structure. Before you start, be sure to take a look at `map.c` and `recurse.c` which form the program. Once you feel familiar with the program, you can compile it by running "`make map`".

Write down the commands you use to complete each step of the following walk-through. Be sure to also **record and submit your answers to all questions in bold to Gradescope**.

a. Run GDB on the `map` executable.

b. Set a breakpoint at the beginning of the program's execution.

c. Run the program until the breakpoint.

d. **What memory address does `argv` store?**

e. **Describe what's located at that memory address. (What does `argv` point to?)**

f. Step until you reach the first call to `recur`.

g. **What is the memory address of the `recur` function?**

h. Step into the first call to `recur`.

i. Step until you reach the `if` statement.

j. Switch into assembly view.

k. Step over instructions until you reach the `callq` instruction.

l. **What values are in all the registers?**

m. Step into the `callq` instruction.

n. Switch back to C code mode.

o. Now print out the current call stack. (Hint: what does the `backtrace` command do?)

p. Now set a breakpoint on the `recur` function which is only triggered when the argument is `0`.

q. Continue until the breakpoint is hit.

r. Print the call stack now.

s. Now go up the call stack until you reach `main`. What was `argc`?

t. Now step until the `return` statement in `recur`.

u. Switch back into the assembly view.

v. **Which instructions correspond to the `return 0` in C?**

w. Now switch back to the source layout.

x. Finish the remaining 3 function calls.

y. Run the program to completion.

z. Quit GDB.

3.4 From Source Code to Executable

Now that you've seen how `map` works, let's take a dive into how we went from high-level C code to an executable.

There are 10 written questions for this section, and you must **submit your responses to these questions on Gradescope.**

Before we start, we'll be using a few compiler flags which are likely new to you. Here's a summary of the flags we'll be using.

- `-Wall` – Enables all compiler warnings
- `-m32` – Compiles the code for the i386 architecture.
- `-S` – Invokes the COMPILER only.
- `-c` – Invokes the COMPILER and ASSEMBLER only.

Let's start by invoking the compiler. The compiler takes high-level C code and produces a variant of x86 known as 8086 or i386 assembly.

To compile `map.c`, run:

```
$ gcc -m32 -S -o map.S map.c
```

This will only invoke the compiler for `map.c` and output the assembly code in `map.S`.

1. Generate `recurse.S` and find which instruction(s) corresponds to the recursive call of `recur(i - 1)`.

Now we will assemble our compiled code into an executable. To assemble our code we can run:

```
$ gcc -m32 -c map.S -o map.o
```

This turns our raw x86 code (`map.S`) into machine code or an object file (`map.o`).
We can also combine these steps by just running `gcc -m32 -c` on our C file directly. We can run:

```
$ gcc -m32 -c recurse.c -o recurse.o
```

The assembler converts the raw assembly code into an object file which contains code as well as other data and metadata necessary for execution. Different operating systems use different types of object files. In this class, we will be using ELF (Executable and Linkable Format), the object format used by Linux. Let's start by taking a look at `map.o` and `recurse.o`. These are object files, so we will use the `objdump` program to read them.

```
$ objdump -D map.o
$ objdump -D recurse.o
```

2. What do the `.text` and `.data` sections contain?

The assembler generates a symbol table which is part of the object file. The symbol table contains all the symbols that can be globally referenced (referenced outside the object file) from another object file (i.e. global/static variables and functions).

3. What command do we use to view the symbols in an ELF file? (Hint: We can use `objdump` again, look at "`man objdump`" to find the right flag).

Here's an excerpt from the `map.o` symbol table:

```
00000000 g     O .data 00000004 stuff
00000000 g     F .text 00000060 main
...
00000000         *UND* 00000000 malloc
00000000         *UND* 00000000 recur
```

4. What do the `g`, `O`, `F`, and `*UND*` flags mean?

5. Where else can we find a symbol for `recur`? Which file is this in? **Copy and paste the relevant portion of the symbol table.**

Finally, let's link our 2 object files to create an executable.

```
$ gcc -m32 map.o recurse.o -o map
```

Note that to we could've just called `gcc -m32 map.c recurse.c -o map` on the C files to do this entire process in a single command. Often times build systems will separate these commands in order to speed up compile times (since only the changed files need to be recompiled).

6. Examine the symbol table of the entire `map` program now. What has changed?

`objdump` can be used to look at more than just the symbol table—it can show us the structure of the executable. Run "`objdump -x -d map`". You will see that your program has several segments, names of functions and variables in your program correspond to labels with addresses or values. 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.

Using the output of `objdump`, answer the following questions:

7. What segment(s)/section(s) contains `recur` (the function)? (The address of `recur` in `objdump` will not be exactly the same as what you saw in `gdb`. An optional stretch exercise is to think about why. See the Wikipedia article on relocation[25] for a hint.)

8. What segment(s)/section(s) contains global variables? Hint: look for the variables `foo` and `stuff`.

9. Do you see the stack segment anywhere? What about the heap? Explain.

10. Based on the output of `map`, in which direction does the stack grow? Explain.

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

3.5 Autograder & Submission

3.5.1 Autograder

To push your code to the autograder do:

```
cd ~/code/personal/hw0
git status
git add limits.c map.c recurse.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 on 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.org/autograder[26] for more information.

Note that the questions with written responses submitted to Gradescope will not be shown in nor graded by the autograder.

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

3.5.2 Gradescope

The written portions of assignments will be submitted to Gradescope. If you're enrolled in the class you should have already been added to Gradescope.

If you're not in Gradescope or the autograder, please make a private post on Piazza and **include your name, email, and Student ID**.

[26] https://cs162.org/autograder