Problem Set 0: Scratch

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Introduce some fundamental programming constructs.
- Empower you to design your own animation, game, or interactive art.
- Impress your friends.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.

- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Important Pre-Course Survey

- If you haven't filled it out already, first take a few minutes to fill out HarvardX's Important Pre-Course Survey, then return here!

Discussion

Reddit

Like to discuss CS50 with classmates? Have a question? Visit CS50's "subreddit" at http://www.reddit.com/r/cs50. You're welcome to browse and search for answers without a reddit account. But if you'd like to ask questions or upvote or downvote posts, you'll want to create a reddit account and log in at https://ssl.reddit.com/login, then return to http://www.reddit.com/r/cs50. That account will be distinct from your edX account, but you're welcome to (try to!) choose the same username.

Be sure to read up on "reddiquette" at http://www.reddit.com/wiki/reddiquette so that you don't get downvoted!

Finish CS50x already? Like to field questions on CS50's subreddit as an alum (i.e., graduate) of CS50? Let us know your reddit username atcs50.net/alum so that we can give you some "flair" so that "alum" appears next to your name when you answer questions, so that students know you're there to help!

Facebook

Prefer to interact with classmates via Facebook? Visit https://www.facebook.com/groups/cs50 and join CS50's Facebook Group!

Twitter

Prefer to interact with classmates via Twitter? Follow @cs50 and tweet with hashtag #cs50!

Bits

- Curl up with Nate's short on binary, if not too familiar:

  And then with Nate's short on ASCII:
- Consider these questions rhetorical for now, but odds are they'll come up again!
  - How do you represent the (decimal) integer 50 in binary?
  - How many bits must be "flipped" (i.e., changed from 0 to 1 or from 1 to 0) in order to capitalize a lowercase `a` that's represented in ASCII?

- How do you represent the (decimal) integer 50 in, oh, "hexadecimal," otherwise known as "base-16"? Know that decimal is considered "base-10" (since it employs 10 digits, 0 through 9), and binary is considered "base-2" (since it employs 2 digits, 0 and 1). Infer from those base systems how to represent base-16! (We'll see base-16 again in the context of graphics and web design.)

- Next dive into Allison's short on Scratch:

  No questions on that one, though, for now!

Itching to Program?

- Now join Zamyla for a walkthrough of this problem set, if you'd like a bit of a tour:
- Next head to http://scratch.mit.edu/ and sign up for an account on MIT's website by clicking **Join Scratch** atop the page. Any username (that's available) is fine, but take care to remember it and your choice of password.
- Then head to http://scratch.mit.edu/help/ and take note of the resources available to you before you dive into Scratch itself. In particular, you might want to skim the Getting Started Guide.
- Next head to http://scratch.mit.edu/projects/266919/ to see **Scratch Scratch Revolution** by CS50 alumna Ann Chi, which Vanessa played on stage in Week 0. Click the blue square above the game's top-left corner if you'd like to full-screen the user interface (UI). Then click either of the green flags. Per Ann's instructions, as soon as you hit your keyboard's spacebar, the game will begin! Feel free to procrastinate a bit. And if you'd like to try out Frogger, by CS50 alumnus Blake Walsh, head to http://scratch.mit.edu/projects/12221773/.
- If you've no experience (or comfort) whatsoever with programming, rest assured that Ann's and Blake's projects are more complex than what we expect for this first problem set. (Click **See inside** in Scratch's top-right corner to look at each project's underlying "implementation details.") But they do reveal what you can do with Scratch. And if your computer has a webcam, you might also want to try**Move the Butterfly** at http://scratch.mit.edu/projects/10016382/. Recall from Week 0 how that program utilizes a webcam to detect movement to which sprites can respond.
- Let's take a look at one other project. Head to http://scratch.mit.edu/projects/37413/ to see a project you probably haven't yet seen by Carlos Herrera. Take a moment to play the game, then click **See inside** in Scratch's top-right corner to see how it's implemented. Spend some time looking over Carlos's scripts. Don't forget that each sprite has its own set of scripts. Try to get a sense of how the overall program works. Try making some changes, even while the program is running, to see how the program responds. Note that this project is probably a bit simpler than we expect of you for this problem set, but it's a good one to learn from because it's pretty easy to follow. And do appreciate that this game, like all Scratch projects, reduces quite literally to some basic building blocks.

  Feel free to download the source code for a few more projects, either from http://scratch.mit.edu/explore/ or from Week 0 athttp://scratch.mit.edu/studios/247678/, even if you already saw some of the latter in lecture. For each program, run it to see how it

works overall and then look over its scripts to understand how it works underneath the hood. Feel free to make changes to scripts and observe the effects. Once you can say to yourself, "Okay, I think I get this," you're ready to proceed.

- Now it's time to choose your own adventure! Your mission is, quite simply, to have fun with Scratch and implement a project of your choice (be it an animation, a game, interactive art, or anything else), subject only to the following requirements.
    - Your project must have at least two sprites, at least one of which must resemble something other than a cat.
    - Your project must have at least three scripts total (i.e., not necessarily three per sprite).
    - Your project must use at least one condition, one loop, and one variable.
    - Your project must use at least one sound.
    - Your project should be more complex than most of those demonstrated in lecture (many of which, though instructive, were quite short) but it can be less complex than, say, Scratch Scratch Revolution. As such, your project should probably use a few dozen puzzle pieces overall.

    Feel free to peruse additional projects online for inspiration, but your own project should not be terribly similar to any of them. Try to think of an idea on your own, and then set out to implement it. But don't try to implement the entirety of your project all at once: pluck off one piece at a time. Ann, for instance, probably implemented just one arrow first, before she moved on to her game's other three. And Carlos probably implemented a stationary goal before he tried to make it move up and down on its own.

    If, along the way, you find it too difficult to implement some feature, try not to fret; alter your design or work around the problem. If you set out to implement an idea that you find fun, you should not find it hard to satisfy this problem set's requirements.

    Alright, off you go. Make us proud!

- Once finished with your project, click **See project page** in Scratch's top-right corner. Ensure your project has a title (in Scratch's top-left corner), some instructions (in Scratch's top-right corner), and some notes and/or credits (in Scratch's bottom-right corner). Then click**Share** in Scratch's top-right corner so that others your project. Finally, take note of the URL in your browser's address bar. That's your project's URL on MIT's website, and you'll need to know it later.
- Oh, and if you'd like to exhibit your project in CS50x's gallery, head to http://scratch.mit.edu/studios/317457/, then click **Add projects**, and paste in your own project's URL.

About You

- Suffice it to say it's a bit harder to meet classmates when taking a course online. But, thanks to technology, everyone can at least say hello!

If you have a digital camera or phone, allow us to invite you to record a 1- to 2-minute video of yourself saying hello to classmates, perhaps stating where in the world you are, why you're taking CS50x, and something interesting about you! In the interests of a fun montage, allow us to suggest that you begin your video by saying "hello, world" and end it with "my name is …., and this is CS50," much like the teaching staff do in CS50's shorts! But, ultimately, it's totally up to you. Indeed, consider this part of the problem set quite optional, but encouraged! We'll do our best to post as many of the videos as possible publicly over time.

If you do record a video, upload it to YouTube (unless blocked in your country, in which case you're welcome to upload it elsewhere) so that you can provide us with its URL when you submit!

CS50 Show

- Have a question about CS50? About computer science? About Harvard? About anything else?

  When you submit this problem set, you'll have an opportunity to submit one or more questions that David and CS50's staff will try to answer for you (and your classmates!) on live, streaming video. (For those unable to tune in live, the videos will also be viewable on demand afterward.) Best would be questions whose answers might appeal to your classmates as well. (For questions about problem sets, the appliance, and the like, best to turn to /r/cs50.)

  If one or more questions do come to mind, record yourself asking them on video, then upload them to YouTube (unless blocked in your country, in which case you're welcome to upload it elsewhere)! Allow us to suggest that you begin your videos by saying something like "Hello, my name is …, and I'm from …" so that we can embed you into the live stream for classmates to see! Alternatively, you're welcome to ask your question simply by typing it out when submitting this problem set, but that'll be less fun! We'll do our best to answer as many questions as possible on camera.

How to Submit

- To submit this problem set, head to the URL below.

  https://x.cs50.net/2014/psets/0/

  You'll find that a few questions await. Be extra-sure that your answers are correct, particularly your email address(es) and your project's URL, else we may overlook your submission! And be sure to click Submit on the last page of that form in order to submit this, your first CS50 problem set!

- This was Problem Set 0.

Problem Set 1: C

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Get comfortable with Linux.
- Start thinking more carefully.
- Solve some problems in C.

Recommended Reading

- Pages 1 – 7, 9, and 10 of http://www.howstuffworks.com/c.htm.
- Chapters 1 – 5, 9, and 11 – 17 of *Absolute Beginner's Guide to C*.
- Chapters 1 – 6 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.

- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Started

- Recall that the CS50 Appliance is a "virtual machine" (running an operating system called Fedora, which itself is a flavor of Linux) that you can run inside of a window on your own computer, whether you run Windows, Mac OS, or even Linux itself. To do so, all you need is a "hypervisor" (otherwise known as a "virtual machine monitor"), software that tricks the appliance into thinking that it's running on "bare metal."

  Alternatively, you could buy a new computer, install Fedora on it (i.e., bare metal), and use that! But a hypervisor lets you do all that for free with whatever computer you already have. Plus, the CS50 Appliance is pre-configured for CS50, so, as soon as you install it, you can hit the ground running.

  So let's get a hypervisor and the CS50 Appliance installed on your computer. Head tohttps://manual.cs50.net/appliance/19/#how_to_install_appliance, where instructions await. In particular, if running Mac OS, follow the instructions for VMware Fusion. If running Windows or Linux, follow the instructions for VMware Player.

- Once you have the CS50 Appliance installed, go ahead and start it (per those same instructions). A small window should open, inside of which the appliance should boot. A few seconds or minutes later, you should find yourself logged in as John Harvard (whose username is**jharvard** and whose password is **crimson**), with John Harvard's desktop before you.

  *If you find that the appliance runs unbearably slow on your PC, particularly if several years old or a somewhat slow netbook, or if you see a hint about "long mode," try the instructions at https://manual.cs50.net/virtualization.

  Feel free to poke around, particularly the 50 Menu in the appliance's bottom-left corner. You should find the graphical user interface (GUI), called Xfce, reminiscent of both Mac OS and Windows. Linux actually comes with a bunch of GUIs; Xfce is just one. If you're already familiar with Linux, you're welcome to install other software via **Menu > Administration > Add/Remove Software**, but the appliance should have everything you need for now. You're also welcome to play with the appliance's various features, per the instructions athttps://manual.cs50.net/appliance/19/#how_to_use_appliance, but this problem set will explicitly mention anything that you need know or do.

- Even if you just downloaded the appliance, ensure that it's completely up-to-date by opening a terminal window, as via **Menu > Programming > Terminal**, typing

```
update50
```

and then hitting Enter on your keyboard. So long as your computer (and, thus, the appliance) has Internet access, the appliance should proceed to download and install any available updates.

- Next, follow the instructions at https://manual.cs50.net/appliance/19/#how_to_synchronize_files_with_dropbox to configure the appliance to use Dropbox so that your work is automatically backed up, just in case something goes wrong with your appliance. (If you really don't want to use Dropbox, that's fine, but realize your files won't be backed up as a result!) If you don't yet have a Dropbox account, sign up when prompted for the free (2 GB) plan. You're welcome to install Dropbox on your own computer as well (outside of the appliance), perhttps://www.dropbox.com/install, but no need if you'd rather not; just inside the appliance is fine.

  If you're already a Dropbox user but don't want your personal files to be synched into the appliance, simply enable **Selective Sync**, per the CS50 Manual's instructions.

- Okay, let's create a folder (otherwise known as a "directory") in which your code for this problem set will soon live. Go ahead and double-click **Home** on John Harvard's desktop (in the appliance's top-left corner). A window entitled **Home** should appear, indicating that you're inside of John Harvard's "home directory" (i.e., personal folder). Then double-click the folder called **Dropbox**, at which point the window's title should change to **Dropbox**. Next select **New Folder** under the gear icon in the window's top-right corner, at which point a new folder called **Untitled Folder** should appear. Rename it **pset1** (in all lowercase, with no spaces). (If the folder's name doesn't seem to be editable, control-click (i.e., click while holding your keyboard's control key) the **Untitled Folder** once, then select **Rename…**, at which point its name should become editable.) Then double-click that **pset1** folder to open it. The window's title should change to **pset1**, and you should see an otherwise empty folder (since you just created it). Notice, though, that atop the window are three buttons, **Home**, **Dropbox**, and **pset1**, that indicate where you were and where you are; you can click buttons like those to navigate back and forth easily.

- Okay, go ahead and close any open windows, then select **Menu > Programming > gedit**. (Recall that Menu is in the appliance's bottom-left corner.) A window entitled **Unsaved Document 1 - gedit** should appear, inside of which is a tab entitled **Unsaved Document 1**. Clearly the document is just begging to be saved. Go ahead and type `hello` (or the ever-popular `asdf`) in the tab, and then notice how the tab's name is now prefixed with an asterisk (*), indicating that you've made changes since the file was first opened. Select **File > Save**, and a window entitled **Save As** should appear. Input `hello.txt` next to **Name**, then click **jharvard** under **Places**. You should then see the contents of John Harvard's home directory. Double-click **Dropbox**, then double-click **pset1**, and you should find yourself inside that empty folder you created. Now, at the bottom of this same window, you should see that the file's default **Character Encoding** is**Unicode (UTF-8)** and that the file's default **Line Ending** is **Unix/Linux**. No need to change either; just notice they're there. That the file's**Line Ending** is **Unix/Linux** just means that `gedit` will insert (invisibly) `\n` at the end of any line of text that you type. Windows, by contrast, uses `\r\n`, and Mac OS uses `\r`, but more on those details some other time.

- Okay, click **Save** in the window's bottom-right corner. The window should close, and you should see that the original window's title is now**hello.txt (~/Dropbox/pset1) - gedit**. The parenthetical just means that **hello.txt** is inside of **pset1**, which is inside of **Dropbox**, which is inside of ~, which is shorthand notation for John Harvard's home directory. A useful reminder is all. The tab, meanwhile, should now be entitled **hello.txt** (with no asterisk, unless you accidentally hit the keyboard again).
- Okay, with `hello.txt` still open in `gedit`, notice that beneath your document is a "terminal window," a command-line (i.e., text-based) interface via which you can navigate the appliance's hard drive and run programs (by typing their name). Notice that the window's "prompt" is

```
jharvard@appliance (~):
```

  which means that you are logged into the appliance as John Harvard and that you are currently inside of ~ (i.e., John Harvard's home directory). If that's the case, there should be a **Dropbox** directory somewhere inside. Let's confirm as much.

  Click somewhere inside of that terminal window, and the prompt should start to blink. Type

```
ls
```

  and then Enter. That's a lowercase L and a lowercase S, which is shorthand notation for "list." Indeed, you should then see a list of the folders inside of John Harvard's home directory, among which is **Dropbox**! Let's open that folder, followed immediately by the **pset1**folder therein. Type

```
cd Dropbox/pset1
```

  or even

```
cd ~/Dropbox/pset1
```

  followed by Enter to change your directory to **~/Dropbox/pset1** (ergo, `cd`). You should find that your prompt changes to

```
jharvard@appliance (~/Dropbox/pset1):
```

  confirming that you are indeed now inside of **~/Dropbox/pset1** (i.e., a directory called **pset1** inside of a directory called **Dropbox** inside of John Harvard's home directory). Now type

```
ls
```

followed by Enter. You should see **hello.txt**! Now, you can't click or double-click on that file's name there; it's just text. But that listing does confirm that hello.txt is where we hoped it would be.

Let's poke around a bit more. Go ahead and type

```
cd
```

and then Enter. If you don't provide `cd` with a "command-line argument" (i.e., a directory's name), it whisks you back to your home directory by default. Indeed, your prompt should now be:

```
jharvard@appliance (~):
```

Phew, home sweet home. Make sense? If not, no worries; it soon will! It's in this terminal window that you'll soon be compiling your first program! For now, though, close `gedit` (via **File > Quit**) and, with it, **hello.txt**.

- Incidentally, if the need arises, know that you can transfer files to and from the appliance per the instructions at https://manual.cs50.net/appliance/19/#how_transfer_files_between_appliance_and_your_computer.

hello, world

- First, a hello from Zamyla if you'd like a tour of what's to come, particularly if less comfortable.
- Shall we have you write your first program? Go ahead and launch `gedit`. (Remember how?) You should find yourself faced with another **Unsaved Document 1**. Go ahead and save the file as `hello.c` (not `hello.txt`) inside of `pset1`, just as before. (Remember how?) Once the file is saved, the window's title should change to **hello.c (~/Dropbox/pset1) - gedit**, and the tab's title should change to **hello.c**. (If either does not, best to close `gedit` and start fresh! Or ask for help!)

  Go ahead and write your first program by typing these lines into the file (though you're welcome to change the words between quotes to whatever you'd like):

```
#include <stdio.h>


int main(void)
{
    printf("hello, world\n");
```

```
}
```

Notice how `gedit` adds "syntax highlighting" (i.e., color) as you type. Those colors aren't actually saved inside of the file itself; they're just added by `gedit` to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, `gedit` wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent.

Do be sure that you type in this program just right, else you're about to experience your first bug! In particular, capitalization matters, so don't accidentally capitalize words (unless they're between those two quotes). And don't overlook that one semicolon. C is quite nitpicky!

When done typing, select **File > Save** (or hit ctrl-s), but don't quit. Recall that the leading asterisk in the tab's name should then disappear. Click anywhere in the terminal window beneath your code, and its prompt should start blinking. But odds are the prompt itself is just

```
jharvard@appliance (~):
```

which means that, so far as the terminal window's concerned, you're still inside of John Harvard's home directory, even though you saved the program you just wrote inside of `~/Dropbox/pset1` (per the top of `gedit`'s window). No problem, go ahead and type

```
cd Dropbox/pset1
```

or

```
cd ~/Dropbox/pset1
```

at the prompt, and the prompt should change to

```
jharvard@appliance (~/Dropbox/pset1):
```

in which case you're where you should be! Let's confirm that `hello.c` is there. Type

```
ls
```

at the prompt followed by Enter, and you should see both `hello.c` and `hello.txt`? If not, no worries; you probably just missed a small step. Best to restart these past several steps or ask for help!

Assuming you indeed see hello.c, let's try to compile! Cross your fingers and then type

```
make hello
```

at the prompt, followed by Enter. (Well, maybe don't cross your fingers whilst typing.) To be clear, type only `hello` here, not `hello.c`. If all that you see is another, identical prompt, that means it worked! Your source code has been translated to 0s and 1s that you can now execute. Type

```
./hello
```

at your prompt, followed by Enter, and you should see whatever message you wrote between quotes in your code! Indeed, if you type

```
ls
```

followed by Enter, you should see a new file, `hello`, alongside `hello.c` and `hello.txt`.

If, though, upon running `make`, you instead see some error(s), it's time to debug! (If the terminal window's too small to see everything, click and drag its top border upward to increase its height.) If you see an error like expected declaration or something no less mysterious, odds are you made a syntax error (i.e., typo) by omitting some character or adding something in the wrong place. Scour your code for any differences vis-à-vis the template above. It's easy to miss the slightest of things when learning to program, so do compare your code against ours character by character; odds are the mistake(s) will jump out! Anytime you make changes to your own code, just remember to re-save via **File > Save** (or ctrl-s), then re-click inside of the terminal window, and then re-type

```
make hello
```

at your prompt, followed by Enter. (Just be sure that you are inside of `~/Dropbox/pset1` within your terminal window, as your prompt will confirm or deny.) If you see no more errors, try running your program by typing

```
./hello
```

at your prompt, followed by Enter! Hopefully you now see precisely the below?

```
hello, world
```

Incidentally, if you find `gedit`'s built-in terminal window too small for your tastes, know that you can open one in its own window via **Menu > Programming > Terminal**. You can then alternate between `gedit` and `Terminal` as needed, as by clicking either's name along the appliance's bottom.

Woo hoo! You've begun to program!

This is CS50 Check

- Now let's see if the program you just wrote is correct! Included in the CS50 Appliance is `check50`, a command-line program with which you can check the correctness of (some of) your programs.

  If not already there, navigate your way to `~/Dropbox/pset1` by executing the command below.

```
cd ~/Dropbox/pset1
```

  If you then execute

```
ls
```

  you should see, at least, `hello.c`. Be sure it's indeed spelled `hello.c` and not `Hello.c`, `hello.C`, or the like. If it's not, know that you can rename a file by executing

```
mv source destination
```

  where `source` is the file's current name, and `destination` is the file's new name. For instance, if you accidentally named your program `Hello.c`, you could fix it as follows.

```
mv Hello.c hello.c
```

  Okay, assuming your file's name is definitely spelled `hello.c` now, go ahead and execute the below. Note that `2014/x/pset1/hello` is just a unique identifier for this problem's checks.

```
check50 2014/x/pset1/hello hello.c
```

  Assuming your program is correct, you should then see output like

```
:) hello.c exists
:) hello.c compiles
:) prints "hello, world\n"
```

  where each green smiley means your program passed a check (i.e., test). You may also see a URL at the bottom of `check50`'s output, but that's just for staff (though you're welcome to visit it).

If you instead see yellow or red smileys, it means your code isn't correct! For instance, suppose you instead see the below.

```
:( hello.c exists
   \ expected hello.c to exist
:| hello.c compiles
   \ can't check until a frown turns upside down
:| prints "hello, world\n"
   \ can't check until a frown turns upside down
```

Because `check50` doesn't think `hello.c` exists, as per the red smiley, odds are you uploaded the wrong file or misnamed your file. The other smileys, meanwhile, are yellow because those checks are dependent on `hello.c` existing, and so they weren't even run.

Suppose instead you see the below.

```
:) hello.c exists
:) hello.c compiles
:( prints "hello, world\n"
   \ expected output, but not "hello, world"
```

Odds are, in this case, you printed something other than `hello, world\n` verbatim, per the spec's expectations. In particular, the above suggests you printed `hello, world`, without a trailing newline (`\n`).

Know that `check50` won't actually record your scores in CS50's gradebook. Rather, it lets you check your work's correctness *before* you submit your work. Once you actually submit your work (per the directions at this spec's end), CS50's staff will use `check50` to evaluate your work's correctness officially.

Shorts

- Head to Week 1's shorts and curl up with Nate's short on libraries. Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!
  - What's a library?
  - What role does

```
#include <cs50.h>
```

play when you write it atop some program?

- What role does

```
-lcs50
```

play when you pass it as a "command-line argument" to `clang`? (Recall that `make`, the program we've been using to compile programs in lecture, simply calls `clang` with some command-line arguments for you to save you some keystrokes.)

- Curl up with at least two other shorts from Week 1. Some additional questions may be in your future!

Itsa Mario

- Toward the end of World 1-1 in Nintendo's Super Mario Brothers, Mario must ascend a "half-pyramid" of blocks before leaping (if he wants to maximize his score) toward a flag pole. Below is a screenshot.

Write, in a file called `mario.c` in your `~/Dropbox/pset1` directory, a program that recreates this half-pyramid using hashes (`#`) for blocks. However, to make things more interesting, first prompt the user for the half-pyramid's height, a non-negative integer no greater than `23`. (The height of the half-pyramid pictured above happens to be `8`.) If the user fails to provide a non-negative integer no greater than `23`, you should re-prompt for the same again. Then, generate (with the help of `printf` and one or more loops) the desired half-pyramid. Take care to align the bottom-left corner of your half-pyramid with the left-hand edge of your terminal window, as in the sample output below, wherein underlined text represents some user's input.

```
jharvard@appliance (~/Dropbox/pset1): ./mario
Height: 8
       ##
      ###
     ####
    #####
   ######
  #######
 ########
#########
```

Note that the rightmost two columns of blocks must be of the same height. No need to generate the pipe, clouds, numbers, text, or Mario himself.

By contrast, if the user fails to provide a non-negative integer no greater than `23`, your program's output should instead resemble the below, wherein underlined text again represents some user's input. (Recall that `GetInt` will handle some, but not all, re-prompting for you.)

```
jharvard@appliance (~/Dropbox/pset1): ./mario
Height: -2
Height: -1
Height: foo
Retry: bar
Retry: 1
##
```

To compile your program, remember that you can execute

```
make mario
```

or, more manually,

```
clang -o mario mario.c -lcs50
```

after which you can run your program with the below.

```
./mario
```

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/pset1/mario mario.c
```

And if you'd like to play with the staff's own implementation of mario in the appliance, you may execute the below.

```
~cs50/pset1/mario
```

Not sure where to begin? Not to worry. A walkthrough awaits!

Time for Change

- Speaking of money, "counting out change is a blast (even though it boosts mathematical skills) with this spring-loaded changer that you wear on your belt to dispense quarters, dimes, nickels, and pennies into your hand." Or so says the website on which we found this here accessory (for ages 5 and up).

Of course, the novelty of this thing quickly wears off, especially when someone pays for a newspaper with a big bill. Fortunately, computer science has given cashiers everywhere ways to minimize numbers of coins due: greedy algorithms.

According to the National Institute of Standards and Technology (NIST), a greedy algorithm is one "that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems."

What's all that mean? Well, suppose that a cashier owes a customer some change and on that cashier's belt are levers that dispense quarters, dimes, nickels, and pennies. Solving this "problem" requires one or more presses of one or more levers. Think of a "greedy" cashier as one who wants to take, with each press, the biggest bite out of this problem as possible. For instance, if some customer is owed 41¢, the biggest first (i.e., best immediate, or local) bite that can be taken is 25¢. (That bite is "best" inasmuch as it gets us closer to 0¢ faster than any other coin would.) Note that a bite of this size would whittle what was a 41¢ problem down to a 16¢ problem, since 41 - 25 = 16. That is, the remainder is a similar but smaller problem. Needless to say, another 25¢ bite would be too big (assuming the cashier prefers not to lose money), and so our greedy cashier would move on to a bite of size 10¢, leaving him or her with a 6¢ problem. At that point, greed calls for one 5¢ bite followed by one 1¢ bite, at which point the problem is solved. The customer receives one quarter, one dime, one nickel, and one penny: four coins in total.

It turns out that this greedy approach (i.e., algorithm) is not only locally optimal but also globally so for America's currency (and also the European Union's). That is, so long as a cashier has enough of each coin, this largest-to-smallest approach will yield the fewest coins possible.

How few? Well, you tell us. Write, in a file called `greedy.c` in your `~/Dropbox/pset1` directory, a program that first asks the user how much change is owed and then spits out the minimum number of coins with which said change can be made. Use `GetFloat` from the CS50 Library to get the user's input and `printf` from the Standard I/O library to output your answer. Assume that the only coins available are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢).

We ask that you use `GetFloat` so that you can handle dollars and cents, albeit sans dollar sign. In other words, if some customer is owed $9.75 (as in the case where a newspaper costs 25¢ but the customer pays with a $10 bill), assume that your program's input will

be `9.75` and not `$9.75` or `975`. However, if some customer is owed $9 exactly, assume that your program's input will be `9.00` or just `9` but, again, not `$9` or `900`. Of course, by nature of floating-point values, your program will likely work with inputs like `9.0` and `9.000` as well; you need not worry about checking whether the user's input is "formatted" like money should be. And you need not try to check whether a user's input is too large to fit in a `float`. But you should check that the user's input makes cents! Er, sense. Using `GetFloat` alone will ensure that the user's input is indeed a floating-point (or integral) value but not that it is non-negative. If the user fails to provide a non-negative value, your program should re-prompt the user for a valid amount again and again until the user complies.

Incidentally, do beware the inherent imprecision of floating-point values. For instance, `0.01` cannot be represented exactly as a float. Try printing its value to, say, `50` decimal places, with code like the below:

```
float f = 0.01;
printf("%.50f\n", f);
```

Before doing any math, then, you'll probably want to convert the user's input entirely to cents (i.e., from a `float` to an `int`) to avoid tiny errors that might otherwise add up! Of course, don't just cast the user's input from a `float` to an `int`! After all, how many cents does one dollar equal? And be careful to round and not truncate your pennies!

Not sure where to begin? Not to worry, start with a walkthrough:

Incidentally, so that we can automate some tests of your code, we ask that your program's last line of output be only the minimum number of coins possible: an integer followed by `\n`. Consider the below representative of how your own program should behave; highlighted in bold is some user's input.

```
jharvard@appliance (~/Dropbox/pset1): ./greedy
O hai!  How much change is owed?
0.41
4
```

By nature of floating-point values, that user could also have inputted just `.41`. (Were they to input `41`, though, they'd get many more coins!)

Of course, more difficult users might experience something more like the below.

```
jharvard@appliance (~/Dropbox/pset1): ./greedy
O hai!  How much change is owed?
```

```
-0.41
How much change is owed?
-0.41
How much change is owed?
foo
Retry: 0.41
4
```

Per these requirements (and the sample above), your code will likely have some sort of loop. If, while testing your program, you find yourself looping forever, know that you can kill your program (i.e., short-circuit its execution) by hitting ctrl-c (sometimes a lot).

We leave it to you to determine how to compile and run this particular program!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/pset1/greedy greedy.c
```

And if you'd like to play with the staff's own implementation of `greedy` in the appliance, you may execute the below.

```
~cs50/pset1/greedy
```

How to Submit

Step 1 of 2

- When ready to submit, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 1** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `hello.c`, as by clicking **jharvard**, then double-clicking **Dropbox**, then double-clicking **pset1**, assuming you saved `hello.c` in `~/Dropbox/pset1`. Once you find `hello.c`, click it once to select it, then click **Open**.
- Click **Add files…** again, and a window entitled **Open Files** should appear again.

- Navigate your way to `mario.c` as before. Click it once to select it, then click **Open**.
- Navigate your way to `greedy.c` as before. Click it once to select it, then click **Open**.
- Click **Start upload** to upload all of your files at once to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50 Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/1/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!
- This was Problem Set 1.

Problem Set 2: Crypto

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Better acquaint you with functions and libraries.
- Allow you to dabble in cryptography.

Recommended Reading

- Pages 11 – 14 and 39 of http://www.howstuffworks.com/c.htm.
- Chapters 6, 7, 10, 17, 19, 21, 22, 30, and 32 of *Absolute Beginner's Guide to C*.
- Chapters 7, 8, and 10 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).

- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.

- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 2's shorts and watch the shorts on loops, scope, and the Caesar Cipher. Then head to Week 3's shorts and watch the short on command-line arguments. Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!
  - How does a while loop differ from a do-while loop? When is the latter particularly useful?
  - What does `undeclared identifier` usually indicate if outputted by `make` (or, really, `clang`)?
  - Why is the Caesar Cipher not very secure?
- And unrelated to those shorts, be sure you're reasonably comfortable answering the below as well when it comes time to submit this problem set's form!
  - What's a function?
  - Why bother writing functions when you can just copy and paste code as needed?

Getting Started

- Alright, here we go again!

  Open a terminal window if not open already (whether by opening gedit via **Menu** > **Programming** > **gedit** or by opening Terminal itself via **Menu** > **Programming** > **Terminal**). Then execute

```
update50
```

  to make sure your appliance is up-to-date. In general, if you run into errors like "No such file or directory" with the staff's solutions or "invalid ID" with `check50`, best to re-run `update50`, just to make sure you have the latest updates.

  Next, execute

```
mkdir ~/Dropbox/pset2
```

  at your prompt in order to make a directory called `pset2` in your `Dropbox` directory. Take care not to overlook the space between `mkdir` and `~/Dropbox/pset2` or any other character for that matter! Keep in mind that `~` denotes your home

directory, `~/Dropbox` denotes a directory called `Dropbox` therein, and `~/Dropbox/pset2` denotes a directory called `pset2` within `~/Dropbox`.

Now execute

```
cd ~/Dropbox/pset2
```

to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
jharvard@appliance (~/Dropbox/pset2):
```

If not, retrace your steps and see if you can determine where you went wrong. You can actually execute

```
history
```

at the prompt to see your last several commands in chronological order if you'd like to do some sleuthing. You can also scroll through the same one line at a time by hitting your keyboard's up and down arrows; hit Enter to re-execute any command that you'd like.

All of the work that you do for this problem set must ultimately reside in your `pset2` directory for submission.

Hail, Caesar!

- Recall from David DiCiurcio's short that Caesar's cipher encrypts (i.e., scrambles in a reversible way) messages by "rotating" each letter by k positions, wrapping around from `Z` to `A` as needed (cf. http://en.wikipedia.org/wiki/Caesar_cipher). In other words, if p is some plaintext (i.e., an unencrypted message), p_i is the i^{th} character in p, and k is a secret key (i.e., a non-negative integer), then each letter, c_i, in the ciphertext, c, is computed as:

  $c_i = (p_i + k) \% 26$

  This formula perhaps makes the cipher seem more complicated than it is, but it's really just a nice way of expressing the algorithm precisely and concisely. And computer scientists love precision and, er, concision.

  For example, suppose that the secret key, k, is 13 and that the plaintext, p, is "Be sure to drink your Ovaltine!" Let's encrypt that p with that k in order to get the ciphertext, c, by rotating each of the letters in p by 13 places, whereby:

  ```
  Be sure to drink your Ovaltine!
  ```

  becomes:

```
Or fher gb qevax lbhe Binygvar!
```

We've deliberately printed the above in a monospaced font so that all of the letters line up nicely. Notice how `O` (the first letter in the ciphertext) is 13 letters away from `B` (the first letter in the plaintext). Similarly is `r` (the second letter in the ciphertext) 13 letters away from `e` (the second letter in the plaintext). Meanwhile, `f` (the third letter in the ciphertext) is 13 letters away from `s` (the third letter in the plaintext), though we had to wrap around from `z` to `a` to get there. And so on. Not the most secure cipher, to be sure, but fun to implement!

Incidentally, a Caesar cipher with a key of 13 is generally called ROT13 (cf. http://en.wikipedia.org/wiki/ROT13). In the real world, though, it's probably best to use ROT26, which is believed to be twice as secure.

Anyhow, your next goal is to write, in `caesar.c`, a program that encrypts messages using Caesar's cipher. Your program must accept a single command-line argument: a non-negative integer. Let's call it *k* for the sake of discussion. If your program is executed without any command-line arguments or with more than one command-line argument, your program should yell at the user and return a value of `1` (which tends to signify an error) immediately as via the statement below:

```
return 1;
```

Otherwise, your program must proceed to prompt the user for a string of plaintext and then output that text with each alphabetical character "rotated" by *k* positions; non-alphabetical characters should be outputted unchanged. After outputting this ciphertext, your program should exit, with main returning `0`.

Although there exist only 26 letters in the English alphabet, you may not assume that *k* will be less than or equal to 26; your program should work for all non-negative integral values of *k* less than $2^{31} - 26$. (In other words, you don't need to worry if your program eventually breaks if the user chooses a value for *k* that's too big or almost too big to fit in an `int`. Now, even if *k* is greater than 26, alphabetical characters in your program's input should remain alphabetical characters in your program's output. For instance, if *k* is 27, `A` should not become `[` even though `[` is 27 positions away from `A` in ASCII; `A` should become `B`, since 27 modulo 26 is 1, as a computer scientists might say. In other words, values like $k = 1$ and $k = 27$ are effectively equivalent.

Your program must preserve case: capitalized letters, though rotated, must remain capitalized letters; lowercase letters, though rotated, must remain lowercase letters.

Where to begin? Well, this program needs to accept a command-line argument, *k*, so this time you'll want to declare `main` with:

```
int main(int argc, string argv[])
```

Recall that `argv` is an "array" of `string`s. You can think of an array as row of gym lockers, inside each of which is some value (and maybe some socks). In this case, inside each such locker is a `string`. To open (i.e., "index into") the first locker, you use syntax like `argv[0]`, since arrays are "zero-indexed." To open the next locker, you use syntax like `argv[1]`. And so on. Of course, if there are `n` lockers, you'd better stop opening lockers once you get to `argv[n - 1]`, since `argv[n]` doesn't exist! (That or it belongs to someone else, in which case you still shouldn't open it.)

And so you can access *k* with code like

```
string k = argv[1];
```

assuming it's actually there! Recall that `argc` is an `int` that equals the number of strings that are in `argv`, so you'd best check the value of argc before opening a locker that might not exist! Ideally, `argc` will be `2`. Why? Well, recall that inside of `argv[0]`, by default, is a program's own name. So `argc` will always be at least `1`. But for this program you want the user to provide a command-line argument, `k`, in which case `argc` should be `2`. Of course, if the user provides more than one command-line argument at the prompt, `argc` could be greater than `2`, in which case it's time for some yelling.

Now, just because the user types an integer at the prompt, that doesn't mean their input will be automatically stored in an `int`. Au contraire, it will be stored as a `string` that just so happens to look like an `int`! And so you'll need to convert that `string` to an actual `int`. As luck would have it, a function, `atoi`, exists for exactly that purposes. Here's how you might use it:

```
int k = atoi(argv[1]);
```

Notice, this time, we've declared `k` as an actual `int` so that you can actually do some arithmetic with it. Ah, much better. Incidentally, you can assume that the user will only type integers at the prompt. You don't have to worry about them typing, say, `foo`, just to be difficult (even though the staff's solution does catch such); `atoi` will just return `0` in such cases.

Because `atoi` is declared in `stdlib.h`, you'll want to `#include` that header file atop your own code. (Technically, your code will compile without it there, since we already `#include` it in `cs50.h`. But best not to trust another library to `#include` header files you know you need.)

Okay, so once you've got `k` stored as an `int`, you'll need to ask the user for some plaintext. Odds are CS50's own `GetString` can help you with that.

Once you have both `k` and some plaintext, it's time to encrypt the latter with the former. Recall that you can iterate over the characters in a string, printing each one at a time, with code like the below:

```
for (int i = 0, n = strlen(p); i < n; i++)
{
    printf("%c", p[i]);
}
```

In other words, just as `argv` is an array of `string`s, so is a `string` an array of `char`s. And so you can use square brackets to access individual characters in `string`s just as you can individual `string`s in `argv`. Neat, eh? Of course, printing each of the characters in a string one at a time isn't exactly cryptography. Well, maybe technically if *k* is 0. But the above should help you help Caesar implement his cipher! For Caesar!

Incidentally, you'll need to `#include` yet another header file in order to use `strlen`.

And Zamyla has some tips for you as well:

So that we can automate some tests of your code, your program must behave per the below. Assumed that the underlined text is what some user has typed.

```
jharvard@appliance (~/Dropbox/pset2): ./caesar 13
Be sure to drink your Ovaltine!
Or fher gb qevax lbhe Binygvar!
```

Besides `atoi`, you might find some handy functions documented at http://www.cs50.net/resources/cppreference.com/stdstring/. For instance, `isdigit` sounds interesting. And, with regard to wrapping around from `Z` to `A` (or `z` to `a`), don't forget about `%`, C's modulo operator. You might also want to check out http://asciitable.com/, which reveals the ASCII codes for more than just alphabetical characters, just in case you find yourself printing some characters accidentally.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/pset2/caesar caesar.c
```

And if you'd like to play with the staff's own implementation of `caesar` in the appliance, you may execute the below.

```
~cs50/pset2/caesar
```

- `uggc://jjj.lbhghor.pbz/jngpu?i=bUt5FWLEUN0`

Parlez-vous français?

- Well that last cipher was hardly secure. Fortunately, per Nate's short on Vigenère's cipher, there's a more sophisticated algorithm out there. Suffice it to say it's French, per http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher. Though do not be mislead by the article's discussion of a tabula recta. Each c_i can be computed with relatively simple arithmetic! You do not need a two-dimensional array.

  Vigenère's cipher improves upon Caesar's by encrypting messages using a sequence of keys (or, put another way, a keyword). In other words, if p is some plaintext and k is a keyword (i.e., an alphbetical string, whereby `A` and `a` represent 0, while `Z` and `z` represent 25), then each letter, c_i, in the ciphertext, c, is computed as:

  $c_i = (p_i + k_j) \% 26$

  Note this cipher's use of k_j as opposed to just k. And recall that, if k is shorter than p, then the letters in k must be reused cyclically as many times as it takes to encrypt p.

  Your final challenge this week is to write, in `vigenere.c`, a program that encrypts messages using Vigenère's cipher. This program must accept a single command-line argument: a keyword, k, composed entirely of alphabetical characters. If your program is executed without any command-line arguments, with more than one command-line argument, or with one command-line argument that contains any non-alphabetical character, your program should complain and exit immediately, with main returning `1` (thereby signifying an error that our own tests can detect). Otherwise, your program must proceed to prompt the user for a string of plaintext, p, which it must then encrypt according to Vigenère's cipher with k, ultimately printing the result and exiting, with `main` returning `0`.

  As for the characters in k, you must treat `A` and `a` as 0, `B` and `b` as 1, … , and `Z` and `z` as 25. In addition, your program must only apply Vigenère's cipher to a character in p if that character is a letter. All other characters (numbers, symbols, spaces, punctuation marks, etc.) must be outputted unchanged. Moreover, if your code is about to apply the j^{th} character of k to the i^{th} character of p, but the latter proves to be a non-alphabetical character, you must wait to apply that j^{th} character of k to the next alphabetical character in p; you must not yet advance to the next character in k. Finally, your program must preserve the case of each letter in p.

Not sure where to begin? As luck would have it, this program's pretty similar to `caesar`! Only this time, you need to decide which character in *k* to use as you iterate from character to character in *p*.

And here's Zamyla again with some tips:

So that we can automate some tests of your code, your program must behave per the below; highlighted in bold are some sample inputs.

```
jharvard@appliance (~/Dropbox/pset2): ./vigenere bacon
Meet me at the park at eleven am
Negh zf av huf pcfx bt gzrwep oz
```

How to test your program, besides predicting what it should output, given some input? Well, recall that we're nice people. And so we've written a program called `devigenere` that also takes one and only one command-line argument (a keyword) but whose job is to take ciphertext as input and produce plaintext as output.

To use our program, execute

```
~cs50/pset2/devigenere k
```

at your prompt, where `k` is some keyword. Presumably you'll want to paste your program's output as input to our program; be sure, of course, to use the same key. Note that you do not need to implement `devigenere` yourself, only `vigenere`.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/pset2/vigenere vigenere.c
```

And if you'd like to play with the staff's own implementation of `vigenere` in the appliance, you may execute the below.

```
~cs50/pset2/vigenere
```

How to Submit

Step 1 of 2

- When ready to submit, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.

- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 2** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `caesar.c`, as by clicking **jharvard**, then double-clicking **Dropbox**, then double-clicking **pset2**. Once you find `caesar.c`, click it once to select it, then click **Open**.
- Click **Add files…** again, and a window entitled **Open Files** should appear again.
- Navigate your way to `vigenere.c` as before. Click it once to select it, then click **Open**.
- Click **Start upload** to upload all of your files at once to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50 Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/2/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 2.

Problem Set 3: Game of Fifteen

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Introduce you to larger programs and programs with multiple source files.
- Accustom you to reading someone else's code.
- Empower you with Makefiles.
- Implement a party favor.

Recommended Reading

- Page 17 of http://www.howstuffworks.com/c.htm.
- Chapters 20 and 23 of *Absolute Beginner's Guide to C*.
- Chapters 13, 15, and 18 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.

- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 3's shorts and watch the shorts on bubble sort, insertion sort, and selection sort. Then head to Week 4's shorts and watch the short on `gdb`. (Phew, so many shorts! And so many sorts! Ha.) Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!
  - `gdb` lets you "debug" program, but, more specifically, what does it let you do?
  - Why does binary search require that an array be sorted?
  - Why is bubble sort in $O(n^2)$?
  - Why is insertion sort in $\Omega(n)$?
  - What's the worst-case running time of merge sort?
  - In no more than 3 sentences, how does selection sort work?

Getting Started

- Recall that, for Problem Sets 1 and 2, you started writing programs from scratch, creating your own `pset1` and `pset2` directories with `mkdir`. For Problem Set 3, you'll instead download "distribution code" (otherwise known as a "distro"), written by us, and add your own lines of code to it. You'll first need to read and understand our code, though, so this problem set is as much about learning to read someone else's code as it is about writing your own!

  Let's get you started. Go ahead and open a terminal window if not open already (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

  ```
  update50
  ```

  to make sure your appliance is up-to-date. Then execute

  ```
  cd ~/Dropbox
  ```

  followed by

```
wget http://cdn.cs50.net/2013/fall/psets/3/pset3/pset3.zip
```

to download a ZIP of this problem set's distro into your appliance (with a command-line program called `wget`). You should see a bunch of output followed by:

```
'pset3.zip' saved
```

If you instead see

```
unable to resolve host address
```

your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

Ultimately, confirm that you've indeed downloaded `pset3.zip` by executing:

```
ls
```

Then, run

```
unzip pset3.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `pset3` as well. Proceed to execute

```
cd pset3
```

followed by

```
ls
```

and you should see that the directory contains two "subdirectories":

```
fifteen  find
```

Fun times ahead!

Searching

- Okay, let's dive into the first of those subdirectories. Execute the command below in a terminal window in your appliance.

```
cd ~/Dropbox/pset3/find
```

  If you list the contents of this directory, you should see the below.

```
helpers.c  helpers.h  Makefile  find.c  generate.c
```

  Wow, that's a lot of files, eh? Not to worry, we'll walk you through them.

- Implemented in `generate.c` is a program that uses a "pseudorandom-number generator" (via a function called `rand`) to generate a whole bunch of random (well, pseudorandom, since computers can't actually generate truly random) numbers, one per line. (Cf.https://www.cs50.net/resources/cppreference.com/stdother/rand.html) Go ahead and compile this program by executing the command below.

```
make generate
```

  Now run the program you just compiled by executing the command below.

```
./generate
```

  You should be informed of the program's proper usage, per the below.

```
Usage: generate n [s]
```

  As this output suggests, this program expects one or two command-line arguments. The first, `n`, is required; it indicates how many pseudorandom numbers you'd like to generate. The second, `s`, is optional, as the brackets are meant to imply; if supplied, it represents the value that the pseudorandom-number generator should use as its "seed." A seed is simply an input to a pseudorandom-number generator that influences its outputs. For instance, if you seed `rand` by first calling `srand` (another function whose purpose is to "seed" `rand`) with an argument of, say, `1`, and then call `rand` itself three times, `rand` might return `17767`, then `9158`, then `39017`. (Cf.https://www.cs50.net/resources/cppreference.com/stdother/srand.html.) But if you instead seed `rand` by first calling `srand` with an argument of, say, `2`, and then call `rand` itself three times, `rand` might instead return `38906`, then `31103`, then `52464`. But if you re-seed `rand` by calling `srand` again with an argument of `1`, the next three times you call `rand`, you'll again get `17767`, then `9158`, then `39017`! See, not so random.

Go ahead and run this program again, this time with a value of, say, `10` for `n`, as in the below; you should see a list of 10 pseudorandom numbers.

```
./generate 10
```

Run the program a third time using that same value for n; you should see a different list of 10 numbers. Now try running the program with a value for s too (e.g., 0), as in the below.

```
./generate 10 0
```

Now run that same command again:

```
./generate 10 0
```

Bet you saw the same "random" sequence of ten numbers again? Yup, that's what happens if you don't vary a pseudorandom number generator's initial seed.

- Now take a look at `generate.c` itself with `gedit`. (Remember how?) Comments atop that file explain the program's overall functionality. But it looks like we forgot to comment the code itself. Read over the code carefully until you understand each line and then comment our code for us, replacing each `TODO` with a phrase that describes the purpose or functionality of the corresponding line(s) of code. (Know that an `unsigned int` is just an `int` that cannot be negative.) And for more details on `rand` and `srand`, recall that you can execute:

- `man rand`

```
man srand
```

Once done commenting `generate.c`, re-compile the program to be sure you didn't break anything by re-executing the command below.

```
make generate
```

If `generate` no longer compiles properly, take a moment to fix what you broke!

Now, recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. Notice, in fact, how `make` just executed a pretty long command for you, per the tool's output. However, as your programs grow in size, `make` won't be able to infer from context anymore how to compile your code; you'll need to start telling `make` how to compile your program,

particularly when they involve multiple source (i.e., `.c`) files. And so we'll start relying on "Makefiles," configuration files that tell `make` exactly what to do.

How did `make` know how to compile generate in this case? It actually used a configuration file that we wrote. Using `gedit`, go ahead and look at the file called `Makefile` that's in the same directory as `generate.c`. This `Makefile` is essentially a list of rules that we wrote for you that tells make how to build generate from `generate.c` for you. The relevant lines appear below.

```
generate: generate.c
    `clang` -ggdb -std=c99 -Wall -Werror -o generate generate.c
```

The first line tells `make` that the "target" called `generate` should be built by invoking the second line's command. Moreover, that first line tells `make` that `generate` is dependent on `generate.c`, the implication of which is that `make` will only re-build generate on subsequent runs if that file was modified since `make` last built `generate`. Neat time-saving trick, eh? In fact, go ahead and execute the command below again, assuming you haven't modified `generate.c`.

```
make generate
```

You should be informed that generate is already up to date. Incidentally, know that the leading whitespace on that second line is not a sequence of spaces but, rather, a tab. Unfortunately, `make` requires that commands be preceded by tabs, so be careful not to change them to spaces with `gedit` (which automatically converts tabs to four spaces), else you may encounter strange errors! The `-Werror` flag, recall, tells `clang` to treat warnings (bad) as though they're errors (worse) so that you're forced (in a good, instructive way!) to fix them.

- Now take a look at `find.c` with `gedit`. Notice that this program expects a single command-line argument: a "needle" to search for in a "haystack" of values. Once done looking over the code, go ahead and compile the program by executing the command below.

```
make find
```

Notice, per that command's output, that `make` actually executed the below for you.

```
clang -ggdb -std=c99 -Wall -Werror -o find find.c helpers.c -lcs50 -
lm
```

Notice further that you just compiled a program comprising not one but two `.c` files: `helpers.c` and `find.c`. How did `make` know what to do? Well, again, open up `Makefile` to see the man behind the curtain. The relevant lines appear below.

```
find: find.c helpers.c helpers.h
    clang -ggdb -std=c99 -Wall -Werror -o find find.c helpers.c -lcs50 -lm
```

Per the dependencies implied above (after the colon), any changes to `find.c`, `helpers.c`, or `helpers.h` will compel `make` to rebuild find the next time it's invoked for this target.

Go ahead and run this program by executing, say, the below.

```
./find 13
```

You'll be prompted to provide some hay (i.e., some integers), one "straw" at a time. As soon as you tire of providing integers, hit ctrl-d to send the program an `EOF` (end-of-file) character. That character will compel `GetInt` from the CS50 Library to return `INT_MAX`, a constant that, per `find.c`, will compel `find` to stop prompting for hay. The program will then look for that needle in the hay you provided, ultimately reporting whether the former was found in the latter. In short, this program searches an array for some value. At least, it should, but it won't find anything yet! That's where you come in. More on your role in a bit.

In turns out you can automate this process of providing hay, though, by "piping" the output of `generate` into `find` as input. For instance, the command below passes 1,000 pseudorandom numbers to `find`, which then searches those values for `42`.

```
./generate 1000 | ./find 42
```

Note that, when piping output from `generate` into `find` in this manner, you won't actually see `generate`'s numbers, but you will see `find`'s prompts.

Alternatively, you can "redirect" `generate`'s output to a file with a command like the below.

```
./generate 1000 > numbers.txt
```

You can then redirect that file's contents as input to `find` with the command below.

```
./find 42 < numbers.txt
```

Let's finish looking at that `Makefile`. Notice the line below.

```
all: find generate
```

This target implies that you can build both `generate` and `find` simply by executing the below.

```
make all
```

Even better, the below is equivalent (because `make` builds a `Makefile`'s first target by default).

```
make
```

If only you could whittle this whole problem set down to a single command! Finally, notice these last lines in `Makefile`:

```
clean:
    rm -f *.o a.out core find generate
```

This target allows you to delete all files ending in `.o` or called `core` (more on that soon!), `find`, or `generate` simply by executing the command below.

```
make clean
```

Be careful not to add, say, `*.c` to that last line in `Makefile`! (Why?) Any line, incidentally, that begins with `#` is just a comment.

- And now the fun begins! Notice that `find.c` calls `search`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully in `helpers.c`! (To be sure, we could have put the contents of `helpers.h` and `helpers.c` in `find.c` itself. But it's sometimes better to organize programs into multiple files, especially when some functions are essentially utility functions that might later prove useful to other programs as well, much like those in the CS50 Library.) Take a peek at `helpers.c` with `gedit`, and you'll see that `search` always returns `false`, whether or not `value` is in `values`. Re-write `search` in such a way that it uses linear search, returning `true` if `value` is in `values` and `false` if `value` is not in `values`. Take care to return `false` right away if `n` isn't even positive.

  When ready to check the correctness of your program, try running the command below.

```
./generate 1000 50 | ./find 2008
```

Because one of the numbers outputted by `generate`, when seeded with `50`, is `2008`, your code should find that "needle"! By contrast, try running the command below as well.

```
./generate 1000 50 | ./find 2013
```

Because `2013` is not among the numbers outputted by `generate`, when seeded with `50`, your code shouldn't find that needle. Best to try some other tests as well, as by running `generate` with some seed, taking a look at its output, then piping that same output to `find`, looking for a "needle" you know to be among the "hay".

Incidentally, note that `main` in `find.c` is written in such a way that `find` returns `0` if the needle is found, else it returns `1`. You can check the so-called "exit code" with which `main` returns by executing

```
echo $?
```

after running some other command. For instance, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2008
echo $?
```

you should see `0`, since `2008` is, again, among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search`(written by you) should return `true`, in which case `main` (written by us) should return (i.e., exit with) `0`. By contrast, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2013
echo $?
```

you should see `1`, since `2013` is, again, not among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search`(written by you) should return `false`, in which case `main` (written by us) should return (i.e., exit with) `1`. Make sense?

When ready to check the correctness of your program officially with `check50`, you may execute the below. Be sure to run the command inside of `~/Dropbox/pset3/find`.

```
check50 2014/x/pset3/find helpers.c
```

Incidentally, be sure not to get into the habit of testing your code with `check50` before testing it yourself. (And definitely don't get into an even worse habit of only testing your code with `check50`!) Suffice it to say `check50` doesn't exist in the real world, so running your code with your own sample inputs, comparing actual output against expected output, is the best habit to get into sooner rather than later. Truly, don't do yourself a long-term disservice!

Anyhow, if you'd like to play with the staff's own implementation of `find` in the appliance, you may execute the below.

```
~cs50/pset3/find
```

Sorting

- Alright, linear search is pretty meh. Recall from Week 0 and Week 3 that we can do better, but first we'd best sort that hay.
- Notice that `find.c` calls `sort`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully too in `helpers.c`! Take a peek at `helpers.c` with `gedit`, and you'll see that `sort` returns immediately, even though `find`'s `main` function does pass it an actual array.

  Now, recall the syntax for declaring an array. Not only do you specify the array's type, you also specify its size between brackets, just as we do for `haystack` in `find.c`:

```
int haystack[MAX];
```

  But when passing an array, you only specify its name, just as we do when passing `haystack` to `sort` in `find.c`:

```
sort(haystack, size);
```

  (Why do we also pass in the size of that array separately?)

  When declaring a function that takes a one-dimensional array as an argument, though, you don't need to specify the array's size, just as we don't when declaring `sort` in `helpers.h` (and `helpers.c`):

```
void sort(int values[], int n);
```

  Go ahead and implement `sort` so that the function actually sorts, from smallest to largest, the array of numbers that it's passed, in such a way that its running time is in $O(n^2)$, where n is the array's size. Odds are you'll want to implement bubble sort, selection sort, or

insertion sort, if only because we discussed them in Week 3. Just realize that there's no one "right" way to implement any of those algorithms; variations abound. In fact, you're welcome to improve upon them as you see fit, so long as your implementation remains in $O(n^2)$. However, take care not to alter our declaration of `sort`. Its prototype must remain:

```
void sort(int values[], int n);
```

As this return type of `void` implies, this function must not return a sorted array; it must instead "destructively" sort the actual array that it's passed by moving around the values therein. As we'll discuss in Week 4, arrays are not passed "by value" but instead "by reference," which means that `sort` will not be passed a copy of an array but, rather, the original array itself.

Although you may not alter our declaration of `sort`, you're welcome to define your own function(s) in `helpers.c` that `sort` itself may then call.

We leave it to you to determine how best to test your implementation of `sort`. But don't forget that `printf` and, per Week 3's first lecture, `gdb` are your friends. And don't forget that you can generate the same sequence of pseudorandom numbers again and again by explicitly specifying `generate`'s seed. Before you ultimately submit, though, be sure to remove any such calls to `printf`, as we like our programs' outputs just they way they are!

Here's Zamyla with some tips:

Incidentally, check out **Resources** on the course's website for a quick-reference guide for `gdb`. If you'd like to play with the staff's own implementation of `find` in the appliance, you may execute the below.

```
~cs50/pset3/find
```

No `check50` for this one!

- Now that `sort` (presumably) works, it's time to improve upon `search`, the other function that lives in `helpers.c`. Recall that your first version implemented linear search. Rip out the lines that you wrote earlier (sniff) and re-implement `search` as Binary Search, that divide-and-conquer strategy that we employed in Week 0 and again in Week 3. You are welcome to take an iterative or, per Week 4, a recursive approach. If you pursue the latter, though, know that you may not change our declaration of `search`, but you may write a new, recursive function (that perhaps takes different parameters) that `search` itself calls. When it comes time to submit this problem set, it suffices to submit this new-and-improved version of `search`; you needn't submit your original version that used linear search.

  Here's Zamyla again:

The Game Begins

- And now it's time to play. The Game of Fifteen is a puzzle played on a square, two-dimensional board with numbered tiles that slide. The goal of this puzzle is to arrange the board's tiles from smallest to largest, left to right, top to bottom, with an empty space in board's bottom-right corner, as in the below.

  Sliding any tile that borders the board's empty space in that space constitutes a "move." Although the configuration above depicts a game already won, notice how the tile numbered 12 or the tile numbered 15 could be slid into the empty space. Tiles may not be moved diagonally, though, or forcibly removed from the board.

  Although other configurations are possible, we shall assume that this game begins with the board's tiles in reverse order, from largest to smallest, left to right, top to bottom, with an empty space in the board's bottom-right corner. If, however, and only if the board contains an odd number of tiles (i.e., the height and width of the board are even), the positions of tiles numbered 1 and 2 must be swapped, as in the below. The puzzle is solvable from this configuration.

- Navigate your way to `~/Dropbox/pset3/fifteen/`, and take a look at `fifteen.c` with `gedit`. (Remember how?) Within this file is an entire framework for the Game of Fifteen. The challenge up next is to complete this game's implementation.

  But first go ahead and compile the framework. (Can you figure out how?) And, even though it's not yet finished, go ahead and run the game. (Can you figure out how?) Odds are you'll want to run it in a separate terminal window, as by opening **Menu > Programming > Terminal**, so that the game fits in your window.

  Phew. It appears that the game is at least partly functional. Granted, it's not much of a game yet. But that's where you come in.

  Read over the code and comments in `fifteen.c` and be sure you can answer the questions below before proceeding further (and again when you submit this problem set's form!).

  - Besides 4 × 4 (which are the Game of Fifteen's dimensions), what other dimensions does the framework allow?
  - With what sort of data structure is the game's board represented?
  - What function is called to greet the player at game's start?
  - What functions do you apparently need to implement?

- Alright, get to it, implement this game. Remember, take "baby steps." Don't try to bite off the entire game at once. Instead, implement one function at a time and be sure that it works before forging ahead. In particular, we suggest that you implement the framework's functions in this order: `init`, `draw`, `move`, `won`. Any design decisions not explicitly prescribed

herein (e.g., how much space you should leave between numbers when printing the board) are intentionally left to you. Presumably the board, when printed, should look something like the below, but we leave it to you to implement your own vision.

```
15 14 13 12

11 10  9  8

 7  6  5  4

```

```
 3  1  2  _
```

Incidentally, recall that the positions of tiles numbered `1` and `2` should only start off swapped (as they are in the 4 × 4 example above) if the board has an odd number of tiles (as does the 4 × 4 example above). If the board has an even number of tiles, those positions should not start off swapped. And so they do not in the 3 × 3 example below:

```
8  7  6

5  4  3

2  1  _
```

Here's Zamyla again if you'd like a hand with `init`:

Or with `draw`:

Or with `move`:

Or with `won`:

- To test your implementation of `fifteen`, you can certainly try playing it. (No `check50` for this one!) Know that you can force your program to quit by hitting ctrl-c. And be sure that you (and we) cannot crash your program, as by providing bogus tile numbers. But know that, much like you automated input into `find`, so can you automate execution of this game. In fact, in `~cs50/pset3` are `3x3.txt` and `4x4.txt`, winning sequences of moves for a 3 × 3 board and a 4 × 4 board, respectively. To test your program with, say, the first of those inputs, execute the below.

```
./fifteen 3 < ~cs50/pset3/3x3.txt
```

Feel free to tweak the appropriate argument to `usleep` to speed up animation. In fact, you're welcome to alter the aesthetics of the game. For (optional) fun with "ANSI escape sequences," including color, take a look at our implementation of `clear` and check outhttp://www.isthe.com/chongo/tech/comp/ansi_escapes.html for more tricks.

You're welcome to write your own functions and even change the prototypes of functions we wrote. But we ask that you not alter the flow of logic in `main` itself so that we can automate some tests of your program once submitted. In particular, `main` must only return `0` if and when the user has actually won the game; non-zero values should be returned in any cases of error, as implied by our distribution code. And be sure not to alter the staff's implementation `save` or `main`'s usage thereof. If in doubt as to whether some design decision of yours might run counter to the staff's wishes, simply contact your teaching fellow.

If you'd like to play with the staff's own solution in the appliance, you may execute the below.

```
~cs50/pset3/fifteen
```

If you'd like to see an even fancier version, one so good that it can play itself, try out our solution to the Hacker Edition by executing the below.

```
~cs50/hacker3/fifteen
```

Instead of typing a number at the game's prompt, type `GOD` instead. Neat, eh?

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset3` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset3`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset3.zip pset3
```

If you type `ls` thereafter, you should see that you have a new file called `pset3.zip` in `~/Dropbox`. (If you realize later that you need to make a change to

some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset3.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 3** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `pset3.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset3.zip`, click it once to select it, then click **Open**.
- Click **Start upload** to upload your ZIP file to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50 Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/3/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 3.

Problem Set 4: Breakout

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

with thanks to Eric Roberts of Stanford

Objectives

- Learn an API.
- Build a game with a real GUI.
- Acquaint you with event handling.
- Impress your friends.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.

- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.

- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 4's shorts and watch the shorts on pointers and strings. Then head to Week 5's shorts and watch the short on the CS50 Library. We may have a few questions for you when it comes time to submit this problem set's form!

Backstory

One day in the late summer of 1975, Nolan Bushnell [founder of Atari and, um, Chuck E. Cheese's], defying the prevailing wisdom that paddle games were over, decided to develop a single-player version of Pong; instead of competing against an opponent, the player would volley the ball into a wall that lost a brick whenever it was hit. He called [Steve] Jobs into his office, sketched it out on his little blackboard, and asked him to design it. There would be a bonus, Bushnell told him, for every chip fewer than fifty that he used. Bushnell knew that Jobs was not a great engineer, but he assumed, correctly, that he would recruit [Steve] Wozniak, who was always hanging around. "I looked at it as a two-for-one thing," Bushnell recalled. "Woz was a better engineer."

Wozniak was thrilled when Jobs asked him to help and proposed splitting the fee. "This was the most wonderful offer in my life, to actually design a game that people would use," he recalled. Jobs said it had to be done in four days and with the fewest chips possible. What he hid from Wozniak was that the deadline was one that Jobs had imposed, because he needed to get to the All One Farm to help prepare for the apple harvest. He also didn't mention that there was a bonus tied to keeping down the number of chips.

"A game like this might take most engineers a few months," Wozniak recalled. "I thought that there was no way I could do it, but Steve made me sure that I could." So he stayed up four nights in a row and did it. During the day at HP, Wozniak would sketch out his design on paper. Then, after a fast-food meal, he would go right to Atari and stay all night. As Wozniak churned out the design, Jobs sat on a bench to his left implementing it by wire-wrapping the chips onto a breadboard. "While Steve was breadboarding, I spent time playing my favorite game ever, which was the auto racing game Gran Trak 10," Wozniak said.

Astonishingly, they were able to get the job done in four days, and Wozniak used only forty-five chips. Recollections differ, but by most accounts Jobs simply gave Wozniak half of the base fee and not the bonus Bushnell paid for saving five chips. It would be another ten years before Wozniak discovered (by being shown the tale in a book on the history of Atari titled *Zap*) that Jobs had been paid this bonus….

Steve Jobs— Walter Isaacson

Getting Started

Your challenge for this problem set is to implement the same game that Steve and Steve did, albeit in software rather than hardware. That game is Breakout.

Now, Problem Set 3 was also a game, but its graphical user interface (GUI) wasn't exactly a GUI; it was more of a textual user interface, since we essentially simulated graphics with `printf`. Let's give Breakout an actual GUI by building atop the Stanford Portable Library (SPL), which is similar in spirit to the CS50 Library but includes an API (application programming interface) for GUI programming and more.

Let's get you started.

- As always, first open a terminal window and execute

```
update50
```

  to make sure your appliance is up-to-date.

- Next execute

```
cd ~/Dropbox
```

  followed by

```
wget http://cdn.cs50.net/2013/fall/lectures/5/m/src5m.zip
```

  to download some source code from Week 5. If you instead see

```
unable to resolve host address
```

  your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

  When ready, unzip the file with

```
unzip src5m.zip
```

  at which point you should find yourself with a directory called `src5m` in `~/Dropbox`. Navigate your way into it with

```
cd src5m
```

  and then execute

```
ls
```

and you should see the below.

```
bounce.c  checkbox.c  cursor.c  Makefile  spl      text.c
button.c  click.c     label.c   slider.c  spl.jar  window.c
```

Go ahead and compile all of these programs at once (thanks to the `Makefile` in there) by executing the below.

```
make
```

Then execute the simplest of those programs as follows.

```
./window
```

A window quite like the below should appear and then disappear after 5 seconds.

Neat, eh? Open up `window.c` with `gedit`. Let's now take a tour.

How did we know how to call `newGWindow` like that? Well, there aren't `man` pages for SPL, but you can peruse the relevant header file (`gwindow.h`). In fact, notice that inside of `src5m` is a subdirectory called `spl`. Inside of that is another subdirectory called `include`. If you take a look there, you'll find `gwindow.h`. Open it up with `gedit` and look around. (Alternatively, you can see it at http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/include/gwindow.h.) Hm, a bit overwhelming. But because SPL's author has commented the code in a standard way, it turns out that you can generate more user-friendly, web-based documentation as a result! Indeed, take a look now at http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gwindow.html, and you'll see a much friendlier format. Click `newGWindow` under **Functions**, and you'll see its prototype:

```
GWindow newGWindow(double width, double height);
```

That's how we knew! See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/index.html for an index into SPL's documentation, though we'll point out more specific places to look.

Now, in the interests of full disclosure, we should mention that SPL is still in beta, so there may be some bugs in its documentation. When in doubt, best to consult those raw header files instead!

- Next open up `click.c` with `gedit`. This one's a bit more involved but it's representative of how to "listen" for "events", quite like those you could "broadcast" in Scratch. Let's take a look.

  See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gevents.html for SPL's documentation of `GEvent`.

- Now open up `cursor.c` with `gedit`. This program, too, handles events, but it also responds to those events by moving a circle (well, a `GOval`) in lockstep. Let's take a look.

  See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gobjects.html for SPL's documentation of `GOval` and other types of objects.

- Next open `bounce.c` with `gedit`. This one uses a bit of arithmetic to bounce a circle back and forth between a window's edges. Let's take a look.

- Now take a look at `button.c`, `checkbox.c`, `label.c`, `slider.c`, and `text.c` in any order with `gedit`. No videos for those, but do try running them each to see what more you can do with SPL.

- Okay, now that you've been exposed to a bit of GUI programming, let's turn our attention to this problem set's own distribution code. In a terminal window, execute

```
cd ~/Dropbox
```

  if not already there, and then execute

```
wget http://cdn.cs50.net/2013/fall/psets/4/pset4/pset4.zip
```

  to download a ZIP of this problem set's distro into your appliance. You should see a bunch of output followed by:

```
'pset4.zip' saved
```

  As before, if you instead see

```
unable to resolve host address
```

  your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

  Ultimately, confirm that you've indeed downloaded `pset4.zip` by executing:

```
ls
```

Then, run

```
unzip pset4.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `pset4` as well. Proceed to execute

```
cd pset4
```

followed by

```
ls
```

and you should see that the directory contains three files and one subdirectory:

- `Makefile`, which will let `make` know how to compile your program;
- `breakout.c`, which contains a skeleton for your program;
- `spl`, a subdirectory containing SPL; and
- `spl.jar`, a "Java archive" that contains code (written in a language called Java) atop which SPL's C library is written.

- Let's see what the distribution code does. Go ahead and execute

```
make breakout
```

or, more simply,

```
make
```

to compile the distro. Then execute

```
./breakout
```

to run the program as is. A window like the below should appear.

Hm, not much of a game. Yet!

- Next try out the staff's solution by executing the below from within your own `~/Dropbox/pset4` directory.

```
~cs50/pset4/breakout
```

  A window like the below should appear.

  How fun! Go ahead and click somewhere inside that window in order to play. The goal, quite simply, is to bounce the ball off of the paddle so as to hit bricks with it. If you break all the bricks, you win! But if you miss the ball three times, you lose! To quit the game, hit control-c back in the terminal window.

  Nice. Let's make your implementation look more like that one. But, first, a tour!

- Open up `breakout.c` with `gedit` and take a moment to scroll through it to get a sense of what lies ahead. It's a bit reminiscent of the skeleton for Game of Fifteen, no? But definitely some new functions in there, most from SPL. Let's walk through it from top to bottom.
  - Atop the file you'll see some familiar header files. We've also included `time.h` so that you have access to a "pseudorandom number generator" (PRNG), a function that can generate random (well, technically not-quite-random) numbers. We've also included some header files from SPL. Because those files are included in this problem set's distribution code (in `pset4/spl/include`), we've used, as is required by C, double quotes (`"`) around their filenames instead of the usual angled brackets (`<` and `>`) because they're not installed deep in the appliance itself.
  - Next up are some constants, values that you don't need to change, but because the code we've written (and that you'll write) needs to know these values in a few places, we've factored them out as constants so that we or you could, theoretically, change them in one convenient location. By contrast, hard-coding the same number (pejoratively known as a "magic number") into your code in multiple places is considered bad practice, since you'd have to remember to change it, potentially, in all of those places.
  - Below those constants are a bunch of prototypes for functions that are defined below `main`. More on each of those soon.
  - Next up is our old friend, `main`. It looks like the first thing that `main` does is "seed" that so-called PRNG with the current time. (See `man srand48` and `man 2 time` if curious.) To seed a PRNG simply means to initialize it in such a way that the numbers it will eventually spit out will appear to be random. It's deliberate, then, that we're initializing the PRNG with the current time: time's always changing. Were we instead to initialize the PRNG with some hard-coded value, it'd always spit out the same sequence of "random" numbers.

    After that call to `srand48`, it looks like `main` calls `newGWindow`, passing in a desired `WIDTH` and `HEIGHT`. That function "instantiates" (i.e., creates) a new graphical

window, returning some sort of reference thereto. (It's technically a pointer, but that detail, and the accompanying `*`, is, again, hidden from us by SPL.) That function's return value is apparently stored in a variable called `window` whose type is `GWindow`, which happens to be declared in a `gwindow.h` header file that you may have glimpsed earlier.

Next, `main` calls `initBricks`, a function written partly by us (and, soon, mostly by you!) that instantiates a grid of bricks atop the game's window.

Then `main` calls `initBall`, which instantiates the ball that will be used to play Breakout. Passed into that function is `window` so that the function knows where to "place" (i.e., draw) the ball. The function returns a `GOval` (graphical oval) whose width and height will simply be equal (ergo a circular ball).

Called by `main` next is `initPaddle`, which instantiates the game's paddle; it returns a `GRect` (graphical rectangle).

Then `main` calls `initScoreboard`, which instantiates the game's scoreboard, which is simply a `GLabel` (graphical label).

Below all those function calls are a few definitions of variables, namely `bricks`, `lives`, and `points`. Below those is a loop, which is meant to iterate again and again so long as the user has lives left to live and bricks left to break. Of course, there's not much code in that loop now!

Below the loop is a call to `waitForClick`, a function that does exactly that so that the window doesn't close until the user intends.

Not too bad, right? Let's next take a closer look at those functions.

- In `initBricks`, you'll eventually write code that instantiates a grid of bricks in the window. Those constants we saw earlier, `ROWS` and `COLS`, represent that grid's dimensions. How to draw a grid of bricks on the screen? Well, odds are you'll want to employ a pair of `for` loops, one nested inside of the other. And within that innermost loop, you'll likely want to instantiate a `GRect` of some width and height (and color!) to represent a brick.
- In `initBall`, you'll eventually write code that instantiates a ball (that is, a circle, or really a `GOval`) and somehow center it in the window.
- In `initPaddle`, you'll eventually write code that instantiates a paddle (just a `GRect`) that's somehow centered in the bottom-middle of the game's window.
- Finally, in `initScoreboard`, you'll eventually write code that instantiates a scoreboard as, quite simply, a `GLabel` whose value is a number (well, technically, a `char*`, which we once knew as a `string`).

- Now, we've already implemented `updateScoreboard` for you. All that function does, given a `GWindow`, a `GLabel`, and an `int`, is convert the `int` to a `string` (okay, `char*`) using a function called `sprintf`, after which it sets the label to that value and then re-centers the label (in case the `int` has more digits than some previous `int`). Why did we allocate an array of size `12` for our representation of that `int` as a `string`? No worries if the reason's non-obvious, but give some though as to how wide the most positive (or most negative!) `int` might be. You're welcome to change this function, but you're not expected to.
- Last up is `detectCollision`, another function that we've written for you. (Phew!) This one's a bit more involved, so do spend some time reading through it. This function's purpose in life, given the ball as a `GOval`, is to determine whether that ball has collided with (i.e., is overlapping) some other object (well, `GObject`) in the game. (A `GRect`, `GOval`, or `GLabel` can also be thought of and treated as a `GObject`, per http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gobjects.html.) To do so, it cuts some corners (figuratively but also kind of literally) by checking whether any of the ball's "corners," as defined by the ball's "bounding box", per the below (wherein *x* and *y* represent coordinates, and *r* represents the ball's radius) are touching some other `GObject` (which might be a brick or a paddle or even something else).

Alright, ready to break out Breakout?

Breakout

Alright, if you're like me, odds are you'll find it easiest to implement Breakout via some baby steps, each of which will get you closer and closer to a great outcome. Rather than try to implement the whole game at once, allow me to suggest that you proceed as follows:

1. Try out the staff's solution again (via `~cs50/pset4/breakout` from within your own `~/Dropbox/pset4` directory) to remind yourself how our implementation behaves. Yours doesn't need to be identical. In fact, all the better if you personalize yours. But playing with our implementation should help guide you toward yours.
2. Implement `initPaddle`. Per the function's return value, your paddle should be implemented as a `GRect`. Odds are you'll first want to decide on a width and height for your paddle, perhaps declaring them both atop `breakout.c` with constants. Then calculate coordinates (*x* and *y*) for your paddle, keeping in mind that it should be initially aligned in the bottom-middle of your game's window. We leave it to you to decide exactly where. Odds are some arithmetic involving the window's width and height and the paddle's width and height will help you center it. Keep in mind that *x* and *y* refer to a `GRect`'s top-left corner, not its own middle. Your paddle's size and location doesn't need to match the staff's

precisely, but it should be perfectly centered, near the window's bottom. You're welcome to choose a color for it too, for which `setColor` and `setFilled` might be of interest. Finally, instantiate your paddle with `newGRect`. (Take note of that function's prototype at http://cdn.cs50.net/2013/fall/psets/4/pset4/pset4/spl/doc/gobjects.html.) Then return the `GRect` returned by `newGRect` (rather than `NULL`, which the distribution code returns only so that the program will compile without `initPaddle` fully implemented).

3. Now, `initPaddle`'s purpose in life is only to instantiate and return a paddle (i.e., `GRect`). It shouldn't handle any of the paddle's movement. For that, turn your attention to the `TODO` up in `main`. Proceed to replace that `TODO` with some lines of code that respond to a user's mouse movements in such a way that the paddle follows the movements, but only along its (horizontal) x-axis. Look back at `cursor.c` for inspiration, but keep in mind that `cursor.c` allowed that circle to move along a (vertical) y-axis as well, which we don't want for Breakout, else the paddle could move anywhere (which might be cool but not exactly Breakout).

4. Now turn your attention to the `TODO` in `initBricks`. Implement that function in such a way that it instantiates a grid of bricks (with `ROWS` rows and `COLS` columns), with each such brick implemented as a `GRect`. Drawing a `GRect` (or even a bunch of them) isn't all that different from drawing a `GOval` (or circle). Odds are, though, you'll want to instantiate them within a `for` loop that's within a `for` loop. (Think back to `mario`, perhaps!) Be sure to leave a bit of a gap between adjacent bricks, just like we did; exactly how many pixels is up to you. And we leave it to you to select your bricks' colors.

5. Now implement `initBall`, whose purpose in life is to instantiate a ball in the window's center. (Another opportunity for a bit of arithmetic!) Per the function's prototype, be sure to return a `GOval`.

6. Then, back in `main`, where there used to be a `TODO`, proceed to write some additional code (within that same `while` loop) that compels that ball to move. Here, too, take baby steps. Look to `bounce.c` first for ideas on how to make the ball bounce back and forth between your window's edges. (Not the ultimate goal, but it's a step toward it!) Then figure out how to make the ball bounce up and down instead of left and right. (Closer!) Then figure out how to make the ball move at an angle. Then, utilize `drand48` to make the ball's initial velocity random, at least along its (horizontal) x-axis. Note that, per its `man` page, `drand48` returns "nonnegative double-precision floating-point values uniformly distributed between [0.0, 1.0)." In other words, it returns a `double` between 0.0 (inclusive) and 1.0 (exclusive). If you want your velocity to be faster than that, simply add some constant to it and/or multiply it by some constant!

   Ultimately, be sure that the ball still bounces off edges, including the window's bottom for now.

7. When ready, add some additional code to `main` (still somewhere inside of that `while` loop) that compels the ball to bounce off of the paddle if it collides with it on its way downward. Odds are you'll want to call that function we wrote, `detectCollision`, inside that loop in order to detect whether the ball's collided with something so that, if so, you can somehow handle such an event. Of course, the ball could collide with the paddle or with any one of those bricks. Keep in mind, then, that `detectCollision` could return any such `GObject`; it's left to you to determine what has been struck. Know, then, that if you store its return value, as with

```
GObject object = detectCollision(window, ball);
```

you can determine whether that `object` is your game's paddle, as with the below.

```
if (object == paddle)
{
    // TODO
}
```

More generally, you can determine if that `object` is a `GRect` with:

```
if (strcmp(getType(object), "GRect") == 0)
{
    // TODO
}
```

Once it comes time to add a `GLabel` to your game (for its scoreboard), you can similarly determine if that `object` is `GLabel`, in which case it might be a collision you want to ignore. (Unless you want your scoreboard to be something the ball can bounce off of. Ours isn't.)

```
if (strcmp(getType(object), "GLabel") == 0)
{
    // TODO
}
```

8. Once you have the ball bouncing off the paddle (and window's edges), focus your attention again on that `while` loop in `main` and figure out how to detect if the ball's hit a brick and

how to remove that brick from the grid if so. Odds are you'll find `removeGWindow` of interest, per http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gwindow.html. **SPL's documentation incorrectly refers to that function as `remove`, but it's indeed `removeGWindow` you want, whose prototype, to be clear, is the below.**

```
void removeGWindow(GWindow gw, GObject gobj);
```

9. Now decide how to determine whether the ball has zoomed past the paddle and struck the window's bottom edge, in which case the user should lose a life and gameplay should probably pause until the user clicks the mouse button, as in the staff's implementation. Odds are detecting this situation isn't all that different from the code you already wrote for bouncing; you just don't want to bounce off that bottom edge anymore!
10. Lastly, implement `initScoreboard` in such a way that the function instantiates and positions a `GLabel` somewhere in your game's window. Then, enhance `main` in such a way that the text of that `GLabel` is updated with the user's score anytime the user breaks a brick. Indeed, be sure that your program keeps track of how many lives remain and how many bricks remain, the latter of which is inversely related to how many points you should give the user for each brick broken; our solution awards one point per brick, but you're welcome to offer different rewards. A user's game should end (i.e., the ball should stop moving) after a user runs out of lives or after all bricks are broken. We leave it to you to decide what to do in both cases, if anything more!

Because this game expects a human to play, no `check50` for this one! Best to invite some friends to find bugs!

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset4` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset4`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset4.zip pset4
```

If you type `ls` thereafter, you should see that you have a new file called `pset4.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset4.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 4** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `pset4.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset4.zip`, click it once to select it, then click **Open**.
- Click **Start upload** to upload your ZIP file to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50 Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/4/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 4.

Problem Set 5: Forensics (Week 7)

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Acquaint you with file I/O.
- Get you more comfortable with data structures, hexadecimal, and pointers.
- Introduce you to computer scientists across campus.
- Help Mr. Boddy.

Recommended Reading*

- Chapters 18, 24, 25, 27, and 28 of *Absolute Beginner's Guide to C*
- Chapters 9, 11, 14, and 16 of *Programming in C*
- http://www.cprogramming.com/tutorial/cfileio.html
- http://en.wikipedia.org/wiki/BMP_file_format
- http://en.wikipedia.org/wiki/Hexadecimal
- http://en.wikipedia.org/wiki/Jpg

* The Wikipedia articles are a bit dense; feel free to skim or skip! ## Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.

- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 7's shorts and watch the shorts on File I/O, Structs, and Valgrind. Just keep in mind that Jason's short on File I/O happens to focus on ASCII (i.e., text) files as opposed to binary files (like images). More on those later!
- You may also want to re-watch the short on GDB among Week 4's shorts!

Getting Started

- Welcome back!
- As always, first open a terminal window and execute

```
update50
```

  to make sure your appliance is up-to-date.

- Like Problem Set 4, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

```
cd ~/Dropbox
```

  in order to navigate to your `~/Dropbox` directory. Then execute

```
wget http://cdn.cs50.net/2013/fall/psets/5/pset5/pset5.zip
```

  in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

  you should see that you now have a file called `pset5.zip` in your `~/Dropbox` directory. Unzip it by executing the below.

```
unzip pset5.zip
```

If you again execute

```
ls
```

you should see that you now also have a `pset5` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f pset5.zip
```

Now dive into that `pset5` directory by executing the below.

```
cd pset5
```

Now execute

```
ls
```

and you should see that the directory contains the below.

```
bmp/  jpg/  questions.txt
```

How fun! Two subdirectories and a file. Who knows what could be inside! Let's get started.

- If you ever saw Windows XP's default wallpaper (think rolling hills and blue skies), then you've seen a BMP. If you've ever looked at a webpage, you've probably seen a GIF. If you've ever looked at a digital photo, you've probably seen a JPEG. If you've ever taken a screenshot on a Mac, you've probably seen a PNG. Read up a bit on the BMP, GIF, JPEG, and PNG file formats. Then, open up `questions.txt` in `~/Dropbox/pset5`, as with `gedit`, and tell us the below.

  1. How many different colors does each format support?
  2. Which of these formats supports animation?
  3. What's the difference between lossy and lossless compression?
  4. Which of these formats is lossy-compressed?

- Curl up with the article from MIT at http://cdn.cs50.net/2013/fall/psets/5/garfinkel.pdf.

  Though somewhat technical, you should find the article's language quite accessible. Once you've read the article, answer each of the following questions in a sentence or more in `~/Dropbox/pset5/questions.txt`.

4. What happens, technically speaking, when a file is deleted on a FAT file system?
5. What can someone like you do to ensure (with high probability) that files you delete cannot be recovered?

Whodunit and more

- Welcome to Tudor Mansion. Your host, Mr. John Boddy, has met an untimely end—he's the victim of foul play. To win this game, you must determine `whodunit`.

  Unfortunately for you (though even more unfortunately for Mr. Boddy), the only evidence you have is a 24-bit BMP file called `clue.bmp`, pictured below, that Mr. Boddy whipped up on his computer in his final moments. Hidden among this file's red "noise" is a drawing of `whodunit`.

  You long ago threw away that piece of red plastic from childhood that would solve this mystery for you, and so you must attack it as a computer scientist instead.

  But, first, some background.

- Perhaps the simplest way to represent an image is with a grid of pixels (i.e., dots), each of which can be of a different color. For black-and-white images, we thus need 1 bit per pixel, as 0 could represent black and 1 could represent white, as in the below. (Image adapted from http://www.brackeen.com/vga/bitmaps.html.)

  In this sense, then, is an image just a bitmap (i.e., a map of bits). For more colorful images, you simply need more bits per pixel. A file format (like GIF) that supports "8-bit color" uses 8 bits per pixel. A file format (like BMP, JPEG, or PNG) that supports "24-bit color" uses 24 bits per pixel. (BMP actually supports 1-, 4-, 8-, 16-, 24-, and 32-bit color.)

  A 24-bit BMP like Mr. Boddy's uses 8 bits to signify the amount of red in a pixel's color, 8 bits to signify the amount of green in a pixel's color, and 8 bits to signify the amount of blue in a pixel's color. If you've ever heard of RGB color, well, there you have it: red, green, blue.

  If the R, G, and B values of some pixel in a BMP are, say, 0xff, 0x00, and 0x00 in hexadecimal, that pixel is purely red, as 0xff (otherwise known as 255 in decimal) implies "a lot of red," while 0x00 and 0x00 imply "no green" and "no blue," respectively. Given how red Mr. Boddy's BMP is, it clearly has a lot of pixels with those RGB values. But it also has a few with other values.

  Incidentally, HTML and CSS (languages in which webpages can be written) model colors in this same way. If curious, seehttp://en.wikipedia.org/wiki/Web_colors for more details.

  Now let's get more technical. Recall that a file is just a sequence of bits, arranged in some fashion. A 24-bit BMP file, then, is essentially just a sequence of bits, (almost) every 24 of which happen to represent some pixel's color. But a BMP file also contains some "metadata,"

information like an image's height and width. That metadata is stored at the beginning of the file in the form of two data structures generally referred to as "headers" (not to be confused with C's header files). (Incidentally, these headers have evolved over time. This problem set only expects that you support version 4.0 (the latest) of Microsoft's BMP format, which debuted with Windows 95.) The first of these headers, called `BITMAPFILEHEADER`, is 14 bytes long. (Recall that 1 byte equals 8 bits.) The second of these headers, called `BITMAPINFOHEADER`, is 40 bytes long. Immediately following these headers is the actual bitmap: an array of bytes, triples of which represent a pixel's color. (In 1-, 4-, and 16-bit BMPs, but not 24- or 32-, there's an additional header right after `BITMAPINFOHEADER` called `RGBQUAD`, an array that defines "intensity values" for each of the colors in a device's palette.) However, BMP stores these triples backwards (i.e., as BGR), with 8 bits for blue, followed by 8 bits for green, followed by 8 bits for red. (Some BMPs also store the entire bitmap backwards, with an image's top row at the end of the BMP file. But we've stored this problem set's BMPs as described herein, with each bitmap's top row first and bottom row last.) In other words, were we to convert the 1-bit smiley above to a 24-bit smiley, substituting red for black, a 24-bit BMP would store this bitmap as follows, where `0000ff` signifies red and `ffffff` signifies white; we've highlighted in red all instances of `0000ff`.

```
ffffff  ffffff  0000ff  0000ff  0000ff  0000ff  ffffff  ffffff
ffffff  0000ff  ffffff  ffffff  ffffff  ffffff  0000ff  ffffff
0000ff  ffffff  0000ff  ffffff  ffffff  0000ff  ffffff  0000ff
0000ff  ffffff  ffffff  ffffff  ffffff  ffffff  ffffff  0000ff
0000ff  ffffff  0000ff  ffffff  ffffff  0000ff  ffffff  0000ff
0000ff  ffffff  ffffff  0000ff  0000ff  ffffff  ffffff  0000ff
ffffff  0000ff  ffffff  ffffff  ffffff  ffffff  0000ff  ffffff
ffffff  ffffff  0000ff  0000ff  0000ff  0000ff  ffffff  ffffff
```

Because we've presented these bits from left to right, top to bottom, in 8 columns, you can actually see the red smiley if you take a step back.

To be clear, recall that a hexadecimal digit represents 4 bits. Accordingly, `ffffff` in hexadecimal actually signifies `111111111111111111111111` in binary.

Okay, stop! Don't proceed further until you're sure you understand why `0000ff` represents a red pixel in a 24-bit BMP file.

- Okay, let's transition from theory to practice. Double-click **Home** on John Harvard's desktop and you should find yourself in John Harvard's home directory. Double-click **pset5**, double-click **bmp**, and then double-click **smiley.bmp** therein. You should see a tiny smiley face that's only 8 pixels by 8 pixels. Select **View > Zoom > Zoom Fit**, and you should see a larger, albeit blurrier, version. (So much for "enhance," huh?) Actually, this particular image shouldn't really be blurry, even when enlarged. The program that launched when you double-clicked**smiley.bmp** (called Ristretto Image Viewer) is simply trying to be helpful (CSI-style) by "dithering" the image (i.e., by smoothing out its edges). Below's what the smiley looks

like if you zoom in without dithering. At this zoom level, you can really see the image's pixels (as big squares).

Okay, go ahead and return your attention to a terminal window, and navigate your way to `~/Dropbox/pset5/bmp`. (Remember how?) Let's look at the underlying bytes that compose `smiley.bmp` using `xxd`, a command-line "hex editor." Execute:

```
xxd -c 24 -g 3 -s 54 smiley.bmp
```

You should see the below; we've again highlighted in red all instances of `0000ff`.

```
0000036: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff
ffffff  ........................
000004e: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff
ffffff  ........................
0000066: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff
0000ff  ........................
000007e: 0000ff ffffff ffffff ffffff ffffff ffffff ffffff
0000ff  ........................
0000096: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff
0000ff  ........................
00000ae: 0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff
0000ff  ........................
00000c6: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff
ffffff  ........................
00000de: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff
ffffff  ........................
```

In the leftmost column above are addresses within the file or, equivalently, offsets from the file's first byte, all of them given in hex. Note that `00000036` in hexadecimal is `54` in decimal. You're thus looking at byte `54` onward of `smiley.bmp`. Recall that a 24-bit BMP's first 14 + 40 = 54 bytes are filled with metadata. If you really want to see that metadata in addition to the bitmap, execute the command below.

```
xxd -c 24 -g 3 smiley.bmp
```

If `smiley.bmp` actually contained ASCII characters, you'd see them in `xxd`'s rightmost column instead of all of those dots.

- So, `smiley.bmp` is 8 pixels wide by 8 pixels tall, and it's a 24-bit BMP (each of whose pixels is represented with 24 ÷ 8 = 3 bytes). Each row (aka "scanline") thus takes up (8 pixels) × (3 bytes per pixel) = 24 bytes, which happens to be a multiple of 4. It turns out that BMPs are stored a bit differently if the number of bytes in a scanline is not, in fact, a multiple

of 4. In `small.bmp`, for instance, is another 24-bit BMP, a green box that's 3 pixels wide by 3 pixels wide. If you view it with Ristretto Image Viewer (as by double-clicking it), you'll see that it resembles the below, albeit much smaller. (Indeed, you might need to zoom in again to see it.)

Each scanline in `small.bmp` thus takes up (3 pixels) ×(3 bytes per pixel) = 9 bytes, which is not a multiple of 4. And so the scanline is "padded" with as many zeroes as it takes to extend the scanline's length to a multiple of 4. In other words, between 0 and 3 bytes of padding are needed for each scanline in a 24-bit BMP. (Understand why?) In the case of small.bmp, 3 bytes' worth of zeroes are needed, since (3 pixels) × (3 bytes per pixel) + (3 bytes of padding) = 12 bytes, which is indeed a multiple of 4.

To "see" this padding, go ahead and run the below.

```
xxd -c 12 -g 3 -s 54 small.bmp
```

Note that we're using a different value for `-c` than we did for `smiley.bmp` so that `xxd` outputs only 4 columns this time (3 for the green box and 1 for the padding). You should see output like the below; we've highlighted in green all instances of `00ff00`.

```
0000036: 00ff00 00ff00 00ff00 000000  ............
0000042: 00ff00 ffffff 00ff00 000000  ............
000004e: 00ff00 00ff00 00ff00 000000  ............
```

For contrast, let's use `xxd` on `large.bmp`, which looks identical to `small.bmp` but, at 12 pixels by 12 pixels, is four times as large. Go ahead and execute the below; you may need to widen your window to avoid wrapping.

```
xxd -c 36 -g 3 -s 54 large.bmp
```

You should see output like the below; we've again highlighted in green all instances of `00ff00`

```
0000036: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00         00ff00         00ff00         00ff00         00ff00
....................................
000005a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00         00ff00         00ff00         00ff00         00ff00
....................................
000007e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00         00ff00         00ff00         00ff00         00ff00
....................................
```

```
00000a2:  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00
00ff00        00ff00        00ff00        00ff00        00ff00
..................................
00000c6:  00ff00  00ff00  00ff00  00ff00  ffffff  ffffff  ffffff
ffffff        00ff00        00ff00        00ff00        00ff00
..................................
00000ea:  00ff00  00ff00  00ff00  00ff00  ffffff  ffffff  ffffff
ffffff        00ff00        00ff00        00ff00        00ff00
..................................
000010e:  00ff00  00ff00  00ff00  00ff00  ffffff  ffffff  ffffff
ffffff        00ff00        00ff00        00ff00        00ff00
..................................
0000132:  00ff00  00ff00  00ff00  00ff00  ffffff  ffffff  ffffff
ffffff        00ff00        00ff00        00ff00        00ff00
..................................
0000156:  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00
00ff00        00ff00        00ff00        00ff00        00ff00
..................................
000017a:  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00
00ff00        00ff00        00ff00        00ff00        00ff00
..................................
000019e:  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00
00ff00        00ff00        00ff00        00ff00        00ff00
..................................
00001c2:  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00  00ff00
00ff00        00ff00        00ff00        00ff00        00ff00
..................................
```

Worthy of note is that this BMP lacks padding! After all, (12 pixels) × (3 bytes per pixel) = 36 bytes is indeed a multiple of 4.

Knowing all this has got to be useful!

- Okay, `xxd` only showed you the bytes in these BMPs. How do we actually get at them programmatically? Well, in `copy.c` is a program whose sole purpose in life is to create a copy of a BMP, piece by piece. Of course, you could just use `cp` for that. But `cp` isn't going to help Mr. Boddy. Let's hope that `copy.c` does!

  Go ahead and compile `copy.c` into a program called `copy` using `make`. (Remember how?) Then execute a command like the below.

```
./copy smiley.bmp copy.bmp
```

  If you then execute ls (with the appropriate switch), you should see that `smiley.bmp` and `copy.bmp` are indeed the same size. Let's double-check that they're actually the same! Execute the below.

```
diff smiley.bmp copy.bmp
```

If that command tells you nothing, the files are indeed identical. (Note that some programs, like Photoshop, include trailing zeroes at the ends of some BMPs. Our version of `copy` throws those away, so don't be too worried if you try to copy a BMP that you've downloaded or made only to find that the copy is actually a few bytes smaller than the original.) Feel free to open both files in Ristretto Image Viewer (as by double-clicking each) to confirm as much visually. But diff does a byte-by-byte comparison, so its eye is probably sharper than yours!

So how now did that copy get made? It turns out that `copy.c` relies on `bmp.h`. Let's take a look. Open up `bmp.h` (as with `gedit`), and you'll see actual definitions of those headers we've mentioned, adapted from Microsoft's own implementations thereof. In addition, that file defines `BYTE`, `DWORD`, `LONG`, and `WORD`, data types normally found in the world of Win32 (i.e., Windows) programming. Notice how they're just aliases for primitives with which you are (hopefully) already familiar. It appears that `BITMAPFILEHEADER` and `BITMAPINFOHEADER` make use of these types. This file also defines a `struct` called `RGBTRIPLE` that, quite simply, "encapsulates" three bytes: one blue, one green, and one red (the order, recall, in which we expect to find RGB triples actually on disk).

Why are these `struct`s useful? Well, recall that a file is just a sequence of bytes (or, ultimately, bits) on disk. But those bytes are generally ordered in such a way that the first few represent something, the next few represent something else, and so on. "File formats" exist because the world has standardized what bytes mean what. Now, we could just read a file from disk into RAM as one big array of bytes. And we could just remember that the byte at location `[i]` represents one thing, while the byte at location `[j]` represents another. But why not give some of those bytes names so that we can retrieve them from memory more easily? That's precisely what the `struct`s in `bmp.h` allow us to do. Rather than think of some file as one long sequence of bytes, we can instead think of it as a sequence of `struct`s.

Recall that `smiley.bmp` is 8 by 8 pixels, and so it should take up $14 + 40 + 8 \cdot 8 \cdot 3 = 246$ bytes on disk. (Confirm as much if you'd like using `ls`.) Here's what it thus looks like on disk according to Microsoft:

As this figure suggests, order does matter when it comes to `struct`s' members. Byte `57 is `rgbtBlue` (and not, say, `rgbtRed`), because `rgbtBlue` is defined first in `RGBTRIPLE`. Our use, incidentally, of the `__attribute__` called `__packed__` ensures that `clang` does not try to "word-align" members (whereby the address of each member's first byte is a multiple of 4), lest we end up with "gaps" in our `struct`s that don't actually exist on disk.

Now go ahead and pull up the URLs to which `BITMAPFILEHEADER` and `BITMAPINFOHEADER` are attributed, per the comments in `bmp.h`. You're about to start using MSDN (Microsoft Developer Network)!

Rather than hold your hand further on a stroll through `copy.c`, we're instead going to ask you some questions and let you teach yourself how the code therein works. As always, `man` is your friend, and so, now, is MSDN. If not sure on first glance how to answer some question, do some quick research and figure it out! You might want to turn to http://www.cs50.net/resources/cppreference.com/stdio/ as well.

Allow us to suggest that you also run `copy` within `gdb` while answering these questions. Set a breakpoint at `main` and walk through the program. Recall that you can tell `gdb` to start running the program with a command like the below at `gdb`'s prompt.

```
run smiley.bmp copy.bmp
```

If you tell `gdb` to print the values of `bf` and `bi` (once read in from disk), you'll see output like the below, which we daresay you'll find quite useful.

```
{bfType = 19778, bfSize = 246, bfReserved1 = 0, bfReserved2 = 0,
  bfOffBits = 54}


{biSize = 40, biWidth = 8, biHeight = -8, biPlanes = 1, biBitCount =
24,
  biCompression = 0, biSizeImage = 192, biXPelsPerMeter = 2834,
  biYPelsPerMeter = 2834, biClrUsed = 0, biClrImportant = 0}
```

In `~/Dropbox/pset5/questions.txt`, answer each of the following questions in a sentence or more.

6. What's `stdint.h`?
7. What's the point of using `uint8_t`, `uint32_t`, `int32_t`, and `uint16_t` in a program?
8. How many bytes is a `BYTE`, a `DWORD`, a `LONG`, and a `WORD`, respectively? (Assume a 32-bit architecture like the CS50 Appliance.)
9. What (in ASCII, decimal, or hexadecimal) must the first two bytes of any BMP file be? (Leading bytes used to identify file formats (with high probability) are generally called "magic numbers.)"
10. What's the difference between `bfSize` and `biSize`?

11. What does it mean if `biHeight` is negative?

12. What field in `BITMAPINFOHEADER` specifies the BMP's color depth (i.e., bits per pixel)?

13. Why might `fopen` return `NULL` in `copy.c:37`?

14. Why is the third argument to `fread` always `1` in our code?

15. What value does `copy.c:70` assign `padding` if `bi.biWidth` is `3`?

16. What does `fseek` do?

17. What is `SEEK_CUR`?

Okay, back to Mr. Boddy.

- Write a program called `whodunit` in a file called `whodunit.c` that reveals Mr. Boddy's drawing.

  Ummm, what?

  Well, think back to childhood when you held that piece of red plastic over similarly hidden messages. (If you remember no such piece of plastic, best to ask a classmate about his or her childhood.) Essentially, the plastic turned everything red but somehow revealed those messages. Implement that same idea in `whodunit`. Like `copy`, your program should accept exactly two command-line arguments. And if you execute a command like the below, stored in `verdict.bmp` should be a BMP in which Mr. Boddy's drawing is no longer covered with noise.

```
./whodunit clue.bmp verdict.bmp
```

  Allow us to suggest that you begin tackling this mystery by executing the command below.

```
cp copy.c whodunit.c
```

  Wink wink. You may be amazed by how few lines of code you actually need to write in order to help Mr. Boddy.

  There's nothing hidden in `smiley.bmp`, but feel free to test your program out on its pixels nonetheless, if only because that BMP is small and you can thus compare it and your own program's output with `xxd` during development. (Or maybe there is a message hidden in `smiley.bmp` too. No, there's not.)

  Rest assured that more than one solution is possible. So long as Mr. Boddy's drawing is identifiable (by you), no matter its legibility, Mr. Boddy will rest in peace.

Because `whodunit` can be implemented in several ways, you won't be able to check your implementation's correctness with `check50`. And, lest it spoil your fun, the staff's solution to `whodunit` is not available.

But here is Zamyla!

- In `~/Dropbox/pset5/questions.txt`, answer the question below. (And yet we used Photoshop.)

  18. Whodunit?

- Well that was fun. Bit late for Mr. Boddy, though.

  Alright, next challenge! Implement now in `resize.c` a program called `resize` that resizes 24-bit uncompressed BMPs by a factor of `n`. Your program should accept exactly three command-line arguments, per the below usage, whereby the first (`n`) must be a positive integer less than or equal to 100, the second the name of the file to be resized, and the third the name of the resized version to be written.

```
Usage: ./resize n infile outfile
```

With a program like this, we could have created `large.bmp` out of `small.bmp` by resizing the latter by a factor of 4 (i.e., by multiplying both its width and it s height by 4), per the below.

```
./resize 4 small.bmp large.bmp
```

You're welcome to get started by copying (yet again) `copy.c` and naming the copy `resize.c`. But spend some time thinking about what it means to resize a BMP. (You may assume that `n` times the size of `infile` will not exceed 2^{32} - 1.) Decide which of the fields in `BITMAPFILEHEADER` and `BITMAPINFOHEADER` you might need to modify. Consider whether or not you'll need to add or subtract padding to scanlines. And be thankful that we don't expect you to support fractional `n` between 0 and 1! (As we do in the Hacker Edition!) But we do expect you to support `n = 1`, the result of which should be an `outfile` with dimensions identical to `infile`'s.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/pset5/resize bmp.h resize.c
```

If you'd like to play with the staff's own implementation of `resize` in the appliance, you may execute the below.

```
~cs50/pset5/resize
```

If you'd like to peek at, e.g., `large.bmp`'s headers (in a more user-friendly way than `xxd` allows), you may execute the below.

```
~cs50/pset5/peek large.bmp
```

Better yet, if you'd like to compare your outfile's headers against the staff's, you might want to execute commands like the below while inside your `~/Dropbox/pset5/bmp` directory. (Think about what each is doing.)

```
./resize 4 small.bmp student.bmp
~cs50/pset5/resize 4 small.bmp staff.bmp
~cs50/pset5/peek student.bmp staff.bmp
```

And here's Zamyla again.

CSI (Computer Science Investigation)

- Alright, now let's put all your new skills to the test.

  In anticipation of this problem set, I spent the past several days snapping photos of people I know, all of which were saved by my digital camera as JPEGs on a 1GB CompactFlash (CF) card. (It's possible I actually spent the past several days on Facebook instead.) Unfortunately, I'm not very good with computers, and I somehow deleted them all! Thankfully, in the computer world, "deleted" tends not to mean "deleted" so much as "forgotten." My computer insists that the CF card is now blank, but I'm pretty sure it's lying to me.

  Write in `~/Dropbox/pset5/jpg/recover.c` a program that recovers these photos.

  Ummm.

  Okay, here's the thing. Even though JPEGs are more complicated than BMPs, JPEGs have "signatures," patterns of bytes that distinguish them from other file formats. In fact, most JPEGs begin with one of two sequences of bytes. Specifically, the first four bytes of most JPEGs are either

  ```
  0xff 0xd8 0xff 0xe0
  ```

  or

  ```
  0xff 0xd8 0xff 0xe1
  ```

from first byte to fourth byte, left to right. Odds are, if you find one of these patterns of bytes on a disk known to store photos (e.g., my CF card), they demark the start of a JPEG. (To be sure, you might encounter these patterns on some disk purely by chance, so data recovery isn't an exact science.)

Fortunately, digital cameras tend to store photographs contiguously on CF cards, whereby each photo is stored immediately after the previously taken photo. Accordingly, the start of a JPEG usually demarks the end of another. However, digital cameras generally initialize CF cards with a FAT file system whose "block size" is 512 bytes (B). The implication is that these cameras only write to those cards in units of 512 B. A photo that's 1 MB (i.e., 1,048,576 B) thus takes up 1048576 ÷ 512 = 2048 "blocks" on a CF card. But so does a photo that's, say, one byte smaller (i.e., 1,048,575 B)! The wasted space on disk is called "slack space." Forensic investigators often look at slack space for remnants of suspicious data.

The implication of all these details is that you, the investigator, can probably write a program that iterates over a copy of my CF card, looking for JPEGs' signatures. Each time you find a signature, you can open a new file for writing and start filling that file with bytes from my CF card, closing that file only once you encounter another signature. Moreover, rather than read my CF card's bytes one at a time, you can read 512 of them at a time into a buffer for efficiency's sake. Thanks to FAT, you can trust that JPEGs' signatures will be "block-aligned." That is, you need only look for those signatures in a block's first four bytes.

Realize, of course, that JPEGs can span contiguous blocks. Otherwise, no JPEG could be larger than 512 B. But the last byte of a JPEG might not fall at the very end of a block. Recall the possibility of slack space. But not to worry. Because this CF card was brand-new when I started snapping photos, odds are it'd been "zeroed" (i.e., filled with 0s) by the manufacturer, in which case any slack space will be filled with 0s. It's okay if those trailing 0s end up in the JPEGs you recover; they should still be viewable.

Now, I only have one CF card, but there are a whole lot of you! And so I've gone ahead and created a "forensic image" of the card, storing its contents, byte after byte, in a file called `card.raw`. So that you don't waste time iterating over millions of 0s unnecessarily, I've only imaged the first few megabytes of the CF card. But you should ultimately find that the image contains 50 JPEGs. As usual, you can open the file programmatically with fopen, as in the below. (It's fine to hard-code this path into your program rather than define it as some constant.)

```
FILE* file = fopen("card.raw", "r");
```

Notice, incidentally, that `~/Dropbox/pset5/jpg` contains only `recover.c`, but it's devoid of any code. (We leave it to you to decide how to implement and compile `recover`!) For simplicity, you should hard-code `"card.raw"` in your program; your program need not accept any command-line arguments. When executed, though, your program should recover every one of the JPEGs from `card.raw`, storing each as a separate file in your current working directory. Your program should number the files it outputs by naming

each `###.jpg`, where `###` is three-digit decimal number from `000` on up. (Befriend `sprintf`.) You need not try to recover the JPEGs' original names. To check whether the JPEGs your program spit out are correct, simply double-click and take a look! If each photo appears intact, your operation was likely a success!

Odds are, though, the JPEGs that the first draft of your code spits out won't be correct. (If you open them up and don't see anything, they're probably not correct!) Execute the command below to delete all JPEGs in your current working directory.

```
rm *.jpg
```

If you'd rather not be prompted to confirm each deletion, execute the command below instead.

```
rm -f *.jpg
```

Just be careful with that `-f` switch, as it "forces" deletion.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/pset5/recover recover.c
```

Lest it spoil your (forensic) fun, the staff's solution to `recover` is not available.

But here's Zamyla!

Sanity Checks

Before you consider this problem set done, best to ask yourself these questions and then go back and improve your code as needed! Do not consider the below an exhaustive list of expectations, though, just some helpful reminders. The checkboxes that have come before these represent the exhaustive list! To be clear, consider the questions below rhetorical. No need to answer them in writing for us, since all of your answers should be "yes!"

- Did you fill `questions.txt` with answers to all questions?
- Is the BMP that `whodunit` outputs legible (to you)?
- Does `resize` accept three and only three command-line arguments?
- Does `resize` ensure that `n` is in [1, 100]?
- Does `resize` update `bfSize`, `biHeight`, `biSizeImage`, and `biWidth` correctly?

- Does `resize` add or remove padding as needed?
- Does `recover` output 50 JPEGs? Are all 50 viewable?
- Does `recover` name the JPEGs `###.jpg`, where `###` is a three-digit number from `000` through `049`?
- Are all of your files where they should be in `~/Dropbox/pset5`?

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset5` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset5`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset5.zip pset5/ -i "*.c" "*.h" "*/Makefile" "*/questions.txt"
```

  If you type `ls` thereafter, you should see that you have a new file called `pset5.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset5.zip`, then create a new ZIP file as before.)
- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 5** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `pset5.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset5.zip`, click it once to select it, then click **Open**.
- Click **Start upload** to upload your ZIP file to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50

Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/5/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 5.

Problem Set 6: Mispellings

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Allow you to design and implement your own data structure.
- Optimize your code's (real-world) running time.

Recommended Reading

- Pages 18 – 20, 27 – 30, 33, 36, and 37 of http://www.howstuffworks.com/c.htm.
- Chapter 26 of *Absolute Beginner's Guide to C*.
- Chapter 17 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).

- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.

- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 7's shorts and watch the shorts on Hash Tables and Tries. You may also want to re-watch the short on Valgrind!

Getting Started

- Start up your appliance and, upon reaching John Harvard's desktop, open a terminal window (remember how?) and execute

```
update50
```

to ensure that your appliance is up-to-date!

- Like Problem Set 5, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

```
cd ~/Dropbox
```

in order to navigate to your `~/Dropbox` directory. Then execute

```
wget http://cdn.cs50.net/2013/fall/psets/6/pset6/pset6.zip
```

in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

you should see that you now have a file called `pset6.zip` in your `~/Dropbox` directory. Unzip it by executing the below.

```
unzip pset6.zip
```

If you again execute

```
ls
```

you should see that you now also have a `pset6` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f pset6.zip
```

Now dive into that `pset6` directory by executing the below.

```
cd pset6
```

Now execute

```
ls
```

and you should see that the directory contains the below.

```
dictionary.c  dictionary.h  Makefile  questions.txt  speller.c
```

Interesting! Let's get started.

- Theoretically, on input of size *n*, an algorithm with a running time of *n* is asymptotically equivalent, in terms of *O*, to an algorithm with a running time of 2*n*. In the real world, though, the fact of the matter is that the latter feels twice as slow as the former.

  The challenge ahead of you is to implement the fastest spell-checker you can! By "fastest," though, we're talking actual, real-world, noticeable seconds—none of that asymptotic stuff this time.

  In `speller.c`, we've put together a program that's designed to spell-check a file after loading a dictionary of words from disk into memory. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both (and a bit more) we leave to you!

  Before we walk you through `speller.c`, go ahead and open up `dictionary.h` with `gedit`. Declared in that file are four functions; take note of what each should do. Now open up `dictionary.c`. Notice that we've implemented those four functions, but only barely, just enough for this code to compile. Your job for this problem set is to re-implement those functions as cleverly as possible so that this spell-checker works as advertised. And fast!

  Let's get you started.

- Recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. However, as your programs grow in size, make won't be able to infer from context anymore how to compile your code; you'll need to start telling make how to compile your program, particularly when they involve multiple source (i.e., `.c`) files, as in the case of this problem set. And so we'll utilize a `Makefile`, a configuration file that tells make exactly what to do. Open up `Makefile` with `gedit`, and let's take a tour of its lines.

The line below defines a variable called `CC` that specifies that make should use `clang` for compiling.

```
CC = clang
```

The line below defines a variable called `CFLAGS` that specifies, in turn, that `clang` should use some flags, most of which should look familiar.

```
CFLAGS = -ggdb3 -O0 -Qunused-arguments -std=c99 -Wall -Werror
```

The line below defines a variable called `EXE`, the value of which will be our program's name.

```
EXE = speller
```

The line below defines a variable called `HDRS`, the value of which is a space-separated list of header files used by `speller`.

```
HDRS = dictionary.h
```

The line below defines a variable called `LIBS`, the value of which is should be a space-separated list of libraries, each of which should be prefixed with `-l`. (Recall our use of `-lcs50` earlier this term.) Odds are you won't need to enumerate any libraries for this problem set, but we've included the variable just in case.

```
LIBS =
```

The line below defines a variable called `SRCS`, the value of which is a space-separated list of C files that will collectively implement speller.

```
SRCS = speller.c dictionary.c
```

The line below defines a variable called `OBJS`, the value of which is identical to that of `SRCS`, except that each file's extension is not `.c` but `.o`.

```
OBJS = $(SRCS:.c=.o)
```

The lines below define a "target" using these variables that tells make how to compile speller.

```
$(EXE): $(OBJS) Makefile
    $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
```

The line below specifies that our `.o` files all "depend on" `dictionary.h` and `Makefile` so that changes to either induce recompilation of the former when you run `make`.

```
$(OBJS): $(HDRS) Makefile
```

Finally, the lines below define another target for cleaning up this problem set's directory.

```
clean:
    rm -f core $(EXE) *.o
```

Know that you're welcome to modify this `Makefile` as you see fit. In fact, you should if you create any `.c` or `.h` files of your own. But be sure not to change any tabs (i.e., `\t`) to spaces, since make expects the former to be present below each target. To be safe, uncheck**Use Spaces** under **Tab Width** at the bottom of `gedit`'s window before modifying `Makefile`.

The net effect of all these lines is that you can compile `speller` with a single command, even though it comprises quite a few files:

```
make speller
```

Even better, you can also just execute:

```
make
```

And if you ever want to delete speller plus any core or .o files, you can do so with a single command:

```
make clean
```

In general, though, anytime you want to compile your code for this problem set, it should suffice to run:

```
make
```

- Okay, next open up `speller.c` with `gedit` and spend some time looking over the code and comments therein. You won't need to change anything in this file, but you should understand it nonetheless. Notice how, by way of `getrusage`, we'll be "benchmarking" (i.e., timing the execution of) your implementations of `check`, `load`, `size`, and `unload`. Also notice how we go about passing `check`, word by word, the contents of some file to be

spell-checked. Ultimately, we report each misspelling in that file along with a bunch of statistics.

Notice, incidentally, that we have defined the usage of `speller` to be

```
Usage: speller [dictionary] text
```

where `dictionary` is assumed to be a file containing a list of lowercase words, one per line, and `text` is a file to be spell-checked. As the brackets suggest, provision of `dictionary` is optional; if this argument is omitted, `speller` will use `/home/cs50/pset6/dictionaries/large` by default. In other words, running

```
./speller text
```

will be equivalent to running

```
./speller ~cs50/pset6/dictionaries/large text
```

where `text` is the file you wish to spell-check. Suffice it to say, the former is easier to type! (Of course, `speller` will not be able to load any dictionaries until you implement `load` in `dictionary.c`! Until then, you'll see **Could not load**.)

Within the default dictionary, mind you, are 143,091 words, all of which must be loaded into memory! In fact, take a peek at that file to get a sense of its structure and size, as with `gedit`. Notice that every word in that file appears in lowercase (even, for simplicity, proper nouns and acronyms). From top to bottom, the file is sorted lexicographically, with only one word per line (each of which ends with `\n`). No word is longer than 45 characters, and no word appears more than once. During development, you may find it helpful to provide `speller` with a `dictionary` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory. In `/home/cs50/pset6/dictionaries/small` is one such dictionary. To use it, execute

```
./speller ~cs50/pset6/dictionaries/small text
```

where `text` is the file you wish to spell-check. Don't move on until you're sure you understand how `speller` itself works!

- Odds are, you didn't spend enough time looking over `speller.c`. Go back one square and walk yourself through it again!
- Okay, technically that last problem induced an infinite loop. But we'll assume you broke out of it. Open up `questions.txt` with `gedit` and answer each of the following questions in one or more sentences.

1. What is pneumonoultramicroscopicsilicovolcanoconiosis?
2. According to its `man` page, what does `getrusage` do?
3. Per that same man page, how many members are in a variable of type `struct rusage`?
4. Why do you think we pass `before` and `after` by reference (instead of by value) to `calculate`, even though we're not changing their contents?
5. Explain as precisely as possible, in a paragraph or more, how `main` goes about reading words from a file. In other words, convince us that you indeed understand how that function's `for` loop works.
6. Why do you think we used `fgetc` to read each word's characters one at a time rather than use `fscanf` with a format string like `"%s"` to read whole words at a time? Put another way, what problems might arise by relying on `fscanf` alone?
7. Why do you think we declared the parameters for `check` and `load` as `const`?

- So that you can test your implementation of `speller`, we've also provided you with a whole bunch of texts, among them the script from *Austin Powers: International Man of Mystery*, a sound bite from Ralph Wiggum, three million bytes from Tolstoy, some excerpts from Machiavelli and Shakespeare, the entirety of the King James V Bible, and more. So that you know what to expect, open and skim each of those files, as with `gedit`. For instance, to open `austinpowers.txt`, open a terminal window and execute the below.

```
gedit ~cs50/pset6/texts/austinpowers.txt
```

  Alternatively, launch `gedit`, select **File** > **Open…**, click **File System** at left, double-click **home** at right, double-click **cs50** at right, double-click **pset6** at right, double-click **texts** at right, then double-click **austinpowers.txt** at right. (If you get lost, simply start these steps over!)

  Now, as you should know from having read over `speller.c` carefully, the output of `speller`, if executed with, say,

```
./speller ~cs50/pset6/texts/austinpowers.txt
```

  will eventually resemble the below. For now, try executing the staff's solution (using the default dictionary) with the below.

```
~cs50/pset6/speller ~cs50/pset6/texts/austinpowers.txt
```

  Below's some of the output you'll see. For amusement's sake, we've excerpted some of our favorite "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

```
MISSPELLED WORDS

[...]
Bigglesworth
[...]
Fembots
[...]
Virtucon
[...]
friggin'
[...]
shagged
[...]
trippy
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

`TIME IN load` represents the number of seconds that `speller` spends executing your implementation of `load`. `TIME IN check` represents the number of seconds that `speller` spends, in total, executing your implementation of `check`. `TIME IN size` represents the number of seconds that `speller` spends executing your implementation of `size`. `TIME IN unload` represents the number of seconds

that `speller` spends executing your implementation of `unload`. `TIME IN TOTAL` is the sum of those four measurements.

Incidentally, to be clear, by "misspelled" we mean that some word is not in the `dictionary` provided. "Fembots" might very well be in some other (swinging) dictionary.

- http://youtu.be/RIevazPIPzU
- Alright, the challenge ahead of you is to implement `load`, `check`, `size`, and `unload` as efficiently as possible, in such a way that `TIME IN load`, `TIME IN check`, `TIME IN size`, and `TIME IN unload` are all minimized. To be sure, it's not obvious what it even means to be minimized, inasmuch as these benchmarks will certainly vary as you feed `speller` different values for `dictionary` and for `text`. But therein lies the challenge, if not the fun, of this problem set. This problem set is your chance to design. Although we invite you to minimize space, your ultimate enemy is time. But before you dive in, some specifications from us.
  - You may not alter `speller.c`.
  - You may alter `dictionary.c` (and, in fact, must in order to complete the implementations of `load`, `check`, `size`, and `unload`), but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
  - You may alter `dictionary.h`, but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
  - You may alter `Makefile`.
  - You may add functions to `dictionary.c` or to files of your own creation so long as all of your code compiles via `make`.
  - Your implementation of `check` must be case-insensitive. In other words, if `foo` is in dictionary, then `check` should return true given any capitalization thereof; none of `foo`, `foO`, `fOo`, `fOO`, `fOO`, `Foo`, `FoO`, `FOo`, and `FOO` should be considered misspelled.
  - Capitalization aside, your implementation of `check` should only return `true` for words actually in `dictionary`. Beware hard-coding common words (e.g., `the`), lest we pass your implementation a `dictionary` without those same words. Moreover, the only possessives allowed are those actually in `dictionary`. In other words, even if `foo` is in `dictionary`, `check` should return `false` given `foo's` if `foo's` is not also in `dictionary`.
  - You may assume that `check` will only be passed strings with alphabetical characters and/or apostrophes.

- You may assume that any `dictionary` passed to your program will be structured exactly like ours, lexicographically sorted from top to bottom with one word per line, each of which ends with `\n`. You may also assume that `dictionary` will contain at least one word, that no word will be longer than `LENGTH` (a constant defined in `dictionary.h`) characters, that no word will appear more than once, and that each word will contain only lowercase alphabetical characters and possibly apostrophes.
- Your spell-checker may only take `text` and, optionally, `dictionary` as input. Although you might be inclined (particularly if among those more comfortable) to "pre-process" our default dictionary in order to derive an "ideal hash function" for it, you may not save the output of any such pre-processing to disk in order to load it back into memory on subsequent runs of your spell-checker in order to gain an advantage.
- You may research hash functions in books or on the Web, so long as you cite the origin of any hash function you integrate into your own code.

  Alright, ready to go?

- Implement `load`!

  Allow us to suggest that you whip up some dictionaries smaller than the 143,091-word default with which to test your code during development. And here's Zamyla with some additional guidance:

- Implement `check`!

  Allow us to suggest that you whip up some small files to spell-check before trying out, oh, War and Peace. And here's Zamyla again:

- Implement `size`!

  If you planned ahead, this one is easy! Here's Zamyla!

- Implement `unload`!

  Be sure to free any memory that you allocated in `load`! Here's Zamyla with some final suggestions!

- In fact, be sure that your spell-checker doesn't leak any memory at all. Recall that `valgrind` is your newest best friend. Know that `valgrind` watches for leaks while your program is actually running, so be sure to provide command-line arguments if you want `valgrind` to analyze `speller` while you use a particular `dictionary` and/or text, as in the below.

```
valgrind --leak-check=full ./speller ~cs50/pset6/texts/austinpowers.txt
```

  If you run `valgrind` without specifying a `text` for `speller`, your implementations of `load` and `unload` won't actually get called (and thus analyzed).

- Don’t forget about your other good buddy, `gdb`.
- And cs50.net/discuss.
- How to check whether your program is outting the right misspelled words? Well, you’re welcome to consult the "answer keys" in `~cs50/pset6/keys`, as with the below.

```
gedit ~cs50/pset6/keys/austinpowers.txt
```

Alternatively, you could run your program on some text in one window, as with the below.

```
./speller ~cs50/pset6/texts/austinpowers.txt
```

And you can then run the staff’s solution on the same text in another window, as with the below.

```
~cs50/pset6/speller ~cs50/pset6/texts/austinpowers.txt
```

And you could then compare the windows visually side by side. That could get tedious quickly, though. So you might instead want to "redirect" your program’s output to a file (just like you may have done with `generate` in Problem Set 3), as with the below.

```
./speller ~cs50/pset6/texts/austinpowers.txt > student.txt
~cs50/pset6/speller ~cs50/pset6/texts/austinpowers.txt > staff.txt
```

You can then compare both files side by side in the same window with a program like `diff`, as with the below.

```
diff -y student.txt staff.txt
```

Alternatively, to save time, you could just compare your program’s output (assuming you redirected it to, e.g., `student.txt`) against one of the answer keys without running the staff’s solution, as with the below.

```
diff -y student.txt ~cs50/pset6/keys/austinpowers.txt
```

If your program’s output matches the staff’s, `diff` will output two columns that should be identical except for, perhaps, the running times at the bottom. If the columns differ, though, you’ll see a `>` or `|` where they differ. For instance, if you see

```
MISSPELLED                                                      WORDS
MISSPELLED WORDS
```

```
FOTTAGE
FOTTAGE
INT                                                             INT

                                                              >
EVIL'S
s                                                               s

                                                              >
EVIL'S
Farbissina
Farbissina
```

that means your program (whose output is on the left) does not think that `EVIL's` is misspelled, even though the staff's output (on the right) does, as is implied by the absence of `EVIL's` in the lefthand column and the presence of `EVIL's` in the righthand column.

To test your code less manually (though still not exhaustively), you may also execute the below:

```
check50 2014/x/pset6/speller dictionary.c dictionary.h Makefile
```

Note that `check50` does not check for memory leaks, so be sure to run `valgrind` as prescribed as well.

- How to assess just how fast (and correct) your code is? Well, as always, feel free to play with the staff's solution, as with the below, and compare its numbers against yours.

```
~cs50/pset6/speller ~cs50/pset6/texts/austinpowers.txt
```

- Congrats! At this point, your speller-checker is presumably complete (and fast!), so it's time for a debriefing. In `questions.txt`, answer each of the following questions in a short paragraph.

7. What data structure(s) did you use to implement your spell-checker? Be sure not to leave your answer at just "hash table," "trie," or the like. Expound on what's inside each of your "nodes."
8. How slow was your code the first time you got it working correctly?
9. What kinds of changes, if any, did you make to your code over the course of the week in order to improve its performance?
10. Do you feel that your code has any bottlenecks that you were not able to chip away at?

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset6` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset6`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset6.zip pset6/ -i "*.c" "*.h" "*/Makefile" "*/questions.txt"
```

  If you type `ls` thereafter, you should see that you have a new file called `pset6.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset6.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 6** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `pset6.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset6.zip`, click it once to select it, then click **Open**.
- Click **Start upload** to upload your ZIP file to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50 Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/6/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 6.

Problem Set 7: C$50 Finance

due 31 December 2014, per schedule

if unable to view embedded YouTube videos, visit CS50.tv for downloadable MP4s and SRTs

Objectives

- Introduce you to HTML, CSS, PHP, and SQL.
- Teach you how to teach yourself new languages.

Recommended Reading

- http://diveintohtml5.info/
- http://php.net/manual/en/langref.php

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.

- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.

- Viewing another's solution to a problem set's problem and basing your own solution on it.

Learning New Languages

Beyond introducing you to web programming, the overarching goal of this problem set is to teach you—nay, empower you—to teach yourself new languages so that you can stand on your own after term's end. We'll guide you through each, but if you nonetheless find yourself Googling and asking lots of questions of classmates and staff, rest assured you're doing it right!

Incidentally, if you'd like to speed up or slow down any of the videos below using Chrome, join YouTube's HTML5 trial athttp://www.youtube.com/html5 and then re-load this specification. In the bottom-right corner of each video (when hovered upon), you should find a gear icon, via which you should be able to choose 2x and more. And once feeling comfortable, you're welcome to skip over any of those videos altogether.

- Let's first take a look at HTTP (Hypertext Transfer Protocol), the protocol via which you can access the web.
- Let's next take a look at HTML (Hypertext Markup Language) in which web pages are written. Here's Daven!
- Let's next take a look at CSS (Cascading Stylesheets) with Joseph.
- So far so good? Let's now leverage HTTP, HTML, and CSS to make our own search engine. Well, at least its front end. Take a peek at**search-0** through **search-4** via the playlist below. (Click **PLAYLIST** in the player's top-left corner for a menu as needed.)
- You are now a web programmer! Okay, not quite. Neither HTML nor CSS are programming languages, but PHP is. Here's Tommy with a look at PHP's syntax. You'll find it's fairly similar to C's!
- Alright, let's now use PHP to make a website with an actual "back end", server-side code that responds to submissions of HTML forms. Take a look at **froshims-0** through **froshims-3** via the playlist below.
- Now let's look at a common "design pattern" for websites called MVC (Model-View-Controller) that we'll ultimately use for this problem set. Take a look at **mvc-0** through **mvc-5** via the playlist below.
- Finally, let's hear about SQL (Structured Query Language). Here's Christopher and cupcakes.
- Phew, bit of a fire hydrant, no? Not to worry, some fun and more comfort await! Let's get you started.

Getting Started

- Start up your appliance and, upon reaching John Harvard's desktop, open a terminal window (remember how?) and execute

```
update50
```

to ensure that your appliance is up-to-date!

- Like Problem Set 6, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

```
cd ~/vhosts
```

in order to navigate to your `~/vhosts` directory. Then execute

```
wget http://cdn.cs50.net/2013/fall/psets/7/pset7/pset7.zip
```

in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

you should see that you now have a file called `pset7.zip` in your `~/vhosts` directory. Unzip it by executing the below.

```
unzip pset7.zip
```

If you again execute

```
ls
```

you should see that you now also have a directory called `pset7`. You're now welcome to delete the ZIP file with the below.

```
rm -f pset7.zip
```

If you next execute

```
cd pset7
```

followed by

```
ls
```

you should see that `pset7` contains three subdirectories: `includes`, `public`, and `templates`. But more on those soon.

- Next, ensure a few directories are world-executable by executing

```
- chmod a+x ~
- chmod a+x ~/vhosts
- chmod a+x ~/vhosts/pset7
```

```
chmod a+x ~/vhosts/pset7/public
```

so that the appliance's web server (and you, from a browser) will be able to access your work. Then, navigate your way to `~/vhosts/pset7/public` by executing the below.

```
cd ~/vhosts/pset7/public
```

If you execute

```
ls
```

you should see that `public` contains four subdirectories and three files. Ensure that the former are word-executable by executing the below.

```
chmod a+x css fonts img js
```

Finally, ensure that the files within those directories are world-readable by executing the below.

```
chmod a+r css/* fonts/* img/* js/*
```

If unfamiliar, `*` is a "wildcard character," so `css/*`, for instance, simply means "all files within the `css` directory."

For security's sake, don't make `~/vhosts/pset7/includes` or `~/vhosts/pset7/templates` world-executable (or their contents world-readable), as they shouldn't be accessible to the whole world (only to your PHP code, as you'll soon see).

- Even though your code for this problem set will live in `~/vhosts/pset7`, let's ensure that it's nonetheless backed up via Dropbox, assuming you set up Dropbox inside of the appliance. In a terminal window, execute

```
ln -s ~/vhosts/pset7 ~/Dropbox
```

in order to create a "symbolic link" (i.e., alias or shortcut) to your `~/vhosts/pset7` directory within your `~/Dropbox` directory so that Dropbox knows to start backing it up.

- So why did we put `pset7` inside of a directory called `vhosts`? Well, the appliance is configured to serve "virtual hosts" (i.e., websites) out of the latter. Specifically, if you visit, say, http://pset7/ using Chrome inside of the appliance, the appliance is configured to look in `~/vhosts/pset7/public` for that website's web-accessible files. But for that to work, we also need to associate the appliance's own IP address with `pset7` so that it "resolves" via DNS to it. Rather than set up a whole DNS server to do that, we can actually edit a file called `hosts` in a directory called `etc`. Let's do that.

  In a terminal window, execute

```
sudo gedit /etc/hosts
```

  in order to run `gedit` as the appliance's "superuser" (aka "root") so that you can edit what's otherwise a read-only file. Carefully add this line at the bottom of that file, which will associate `pset7` with the appliance's "loopback" address (which won't ever change):

```
127.0.0.1 pset7
```

  Then save the file and quit `gedit`. Then enjoy http://xkcd.com/149/.

  If you encounter a **Gtk-WARNING** with `gedit`, try restarting the appliance, as via **Menu > Log Out > Restart** (or via VMware's own **Restart** option), then try again; the warning appears to be a bug in Fedora (the appliance's operating system).

- Alright, time for a test! Open up Chrome inside of the appliance and visit http://pset7/.

  You should find yourself redirected to C$50 Finance! (If you instead see Forbidden, odds are you missed a step earlier; best to try all those chmod steps again.) If you try logging into C$50 Finance with a username of, oh, **skroob** and a password of **12345**, you should encounter an error about an **Unknown database**. That's simply because you haven't created it yet! Let's create it.

  Head to http://pset7/phpMyAdmin using Chrome inside of the appliance to access phpMyAdmin, a Web-based tool (that happens to be written in PHP) with which you can manage MySQL databases. (MySQL is a free, open-source database that CS50, Facebook, and lots of other sites use.) Log in as John Harvard if prompted (with a username of **jharvard** and a password of **crimson**). You should then find yourself at phpMyAdmin's main page. In phpMyAdmin's top-left corner, you should see **No databases**. Normally, you

can create a database by clicking phpMyAdmin's **Databases** tab, but you can also execute some SQL commands manually.

Go ahead and visit http://cdn.cs50.net/2013/fall/psets/7/pset7/pset7.sql using Chrome inside of the appliance, and then open the file in `gedit`, as by clicking its name in Chrome's bottom-left corner or by selecting **File > Open…** in `gedit` and then navigating your way to**Downloads**. You should ultimately see a whole bunch of SQL (i.e., database queries) within `pset7.sql`. Highlight it all, then select **Edit > Copy** (or hit ctrl-c), then return to phpMyAdmin. Click phpMyAdmin's **SQL** tab, and paste everything you copied into that page's big text box. Skim what you just pasted to get a sense of the commands you're about to execute, then click **Go**. You should then see a green banner, proclaiming **Your SQL query has been executed successfully**. In phpMyAdmin's top-left corner, you should now see link to a database called **pset7**, beneath which is a link to a table called **users**. But more on those later.

Return to http://pset7/ using Chrome inside of the appliance and reload that page. Then try to log in with a username of **skroob** and a password of **12345**. This time, you should see some construction.

- Okay, time for a heads-up. Anytime you create a new file or directory in `~/vhosts/pset7` or some subdirectory therein for this problem set, you'll want to set its permissions with `chmod`. Thus far, we've relied on `a+r` and `a+x`, but let's empower you with more precise control over permissions.

  Henceforth, for any PHP file, file, that you create, execute

```
chmod 600 file
```

  so that it's accessible only by you (and the appliance's webserver). After all, we don't want visitors to see the contents of PHP files; rather, we want them to see the output of PHP files once executed (or, rather, interpreted) by the appliance's web server.

  For any non-PHP file, file, that you create (or upload), execute

```
chmod 644 file
```

  so that it's accessible via a browser (if that's indeed your intention).

  And for any directory, directory, that you create, execute

```
chmod 711 directory
```

  so that its contents are accessible via a browser (if that's indeed your intention).

  What's with all these numbers we're having you type? Well, `600` happens to mean `rw-------`, and so all PHP files are made readable and writable only by you; `644` happens to

mean `rw-r--r--`, and so all non-PHP files are to be readable and writable by you and just readable by everyone else; and `711` happens to mean `rwx--x--x`, and so all directories are to be readable, writable, and executable by you and just executable by everyone else. Wait a minute, don't we want everyone to be able to read (i.e., interpret) your PHP files? Nope! For security reasons, PHP-based web pages are interpreted "as you" (i.e., under John Harvard's username) in the appliance. For the curious, we're using suPHP with Apache.

Okay, still, what's with all those numbers? Well, think of `rw-r--r--` as representing three triples of bits, the first triple of which, to be clear, is `rw-`. Imagine that `-` represents `0`, whereas `r`, `w`, and `x` represent `1`. And, so, this same triple (`rw-`) is just `110` in binary, or `6` in decimal! The other two triples, `r--` and `r--`, then, are just `100` and `100` in binary, or `4` and `4` in decimal! How, then, to express a pattern like `rw-r--r--` with numbers? Why, with `644`.

Actually, this is a bit of a white lie. Because you can represent only eight possible values with three bits, these numbers (`6`, `4`, and `4`) are not actually decimal digits but "octal." So you can now tell your friends that you speak not only binary, decimal, and hexadecimal, but octal as well.

Yahoo!

- If you're not quite sure what it means to buy and sell stocks (i.e., shares of a company), surf on over tohttp://www.investopedia.com/university/stocks/ for a tutorial.

  You're about to implement C$50 Finance, a Web-based tool with which you can manage portfolios of stocks. Not only will this tool allow you to check real stocks' actual prices and portfolios' values, it will also let you buy (okay, "buy") and sell (fine, "sell") stocks! (Per Yahoo's fine print, "Quotes delayed [by a few minutes], except where indicated otherwise.")

- Just the other day, I received the stock tip below in my inbox!

  "Discovery Ventures Signs Letter Of Intent To Acquire The Willa Gold Deposit"

  Let's get in on this opportunity now. Head on over to Yahoo! Finance at http://finance.yahoo.com/. Type the symbol for Discovery Ventures,**DVN.V**, into the text field in that page's top-left corner and click **Get Quotes**. Odds are you'll see a table like the below, which no one has apparently yet Liked!

  Wow, only 22.5 cents per share! That must be a good thing. Anyhow, notice how Yahoo reports a stock's most recent (i.e., "Last Trade") price ($0.27) and more. Moreover, scroll down to the page's bottom, and you should see a toolbox like the below.

  Looks like Yahoo lets you download all that data. Go ahead and click **Download Data** to download a file in CSV format (i.e., as comma-separated values). Open the file in Excel or

any text editor (e.g., `gedit`), and you should see a "row" of values, all excerpted from that table. It turns out that the link you just clicked led to the URL below.

http://download.finance.yahoo.com/d/quotes.csv?s=DVN.V&f=sl1d1t1c1ohgv&e=.csv

Notice how Discovery Ventures' symbol is embedded in this URL (as the value of the HTTP parameter called `s`); that's how Yahoo knows whose data to return. Notice also the value of the HTTP parameter called `f`; it's a bit cryptic (and officially undocumented), but the value of that parameter tells Yahoo which fields of data to return to you. If curious as to what they mean, head to http://www.gummy-stuff.org/Yahoo-data.htm.

It's worth noting that a lot of websites that integrate data from other websites do so via "screen scraping," a process that requires writing programs that parse (or, really, search) HTML for data of interest (e.g., air fares, stock prices, etc.). Writing a screen scraper for a site tends to be a nightmare, though, because a site's markup is often a mess, and if the site changes the format of its pages overnight, you need to re-write your scraper. (See https://manual.cs50.net/scraping/ if curious as to how it can be done nonetheless.)

Thankfully, because Yahoo provides data in CSV, C$50 Finance will avoid screen scraping altogether by downloading (effectively pretending to be a browser) and parsing CSV files instead. Even more thankfully, we've written that code for you!

In fact, let's turn our attention to the code you've been given.

- Navigate your way to `~/vhosts/pset7/public` and open up `index.php` with `gedit`. (Remember how?) Know that `index.php` is the file that's loaded by default when you visit a URL like http://pset7/. Well, it turns out there's not much PHP code in this file. And there isn't any HTML at all. Rather, `index.php` "requires" `config.php` (which is in a directory called `includes` in `index.php`'s parent directory). And `index.php` then calls `render` (a function implemented in a file called `functions.php` that can also be found inside of `includes`) in order to render (i.e., output) a template called `portfolio.php` (which is in a directory called `templates` in `index.php`'s parent directory). Phew, that was a mouthful.

  It turns out that `index.php` is considered a "controller," whereby its purpose in life is to control the behavior of your website when a user visits http://pset7/ (or, equivalently, http://pset7/index.php). Eventually, you'll need to add some more PHP code to this file in order to pass more than just title to render. But for now, let's take a look at `portfolio.php`, the template that this controller ultimately renders.

  Navigate your way to `~/vhosts/pset7/templates` and open up `portfolio.php` with `gedit`. Ah, there's some HTML. Of course, it's not very interesting HTML, but it does explain why your website is "under construction," thanks to the GIF referenced therein.

  Now navigate your way to `~/vhosts/pset7/includes` and open up `config.php` with `gedit`. Recall that `config.php` was required by `index.php`. Notice

how `config.php` first enables display of all errors (and warnings and notices, which are less severe errors) so that you're aware of any syntactical mistakes (and more) in your code. Notice, too, that `config.php` itself requires two other files: `constants.php` and `functions.php`. Next, `config.php` calls `session_start` in order to enable `$_SESSION`, a "superglobal" variable via which we'll remember that a user is logged in. (Even though HTTP is a "stateless" protocol, whereby browsers are supposed to disconnect from servers as soon as they're done downloading pages, "cookies" allow browsers to remind servers who they or, really, you are on subsequent requests for content. PHP uses "session cookies" to provide you with `$_SESSION`, an associative array in which you can store any data to which you'd like to have access for the duration of some user's visit. The moment a user ends his or her "session" (i.e., visit) by quitting his or her browser, the contents of `$_SESSION` are lost for that user specifically because the next time that user visits, he or she will be assigned a new cookie!) Meanwhile, `config.php` uses a "regular expression" (via a call to `preg_match`) to redirect the users to `login.php` anytime they visit some page other than `login.php`, `logout.php`, and `register.php`, assuming `$_SESSION["id"]`isn't yet set. In other words, that block of code requires users to log in if they aren't logged in already (and if they aren't already at one of those three pages).

Okay, now open up `functions.php` with `gedit`. Interesting, it looks like `functions.php` requires `constants.php`. More on that file, though, in a moment. It looks like `functions.php` also defines a bunch of functions, the first of which is `apologize`, which you can call anytime you need to apologize to the user (because they made some mistake). Defined next is `dump`, which you're welcome to call anytime you want to see the contents (perhaps recursively) of some variable while developing your site. That function is only for diagnostic purposes, though. Be sure to remove all calls thereto before submitting your work. Next in the file is `logout`, a function that logs users out by destroying their sessions. Thereafter is `lookup`, a function that queries Yahoo Finance for stocks' prices and more. More on that, though, in a bit. Up next is `query`, a function that executes a SQL query and then returns the result set's rows, if any. Below it is `redirect`, a function that allows you to redirect users from one URL to another. Last in the file is `render`, the function that `index.php` called in order to render `portfolio.php`. The function then "extracts" those values into the local scope (whereby a key of `"foo"` with a value of `"bar"` in `$values` becomes a local variable called `$foo` with a value of `"bar"`). And it then requires `header.php` followed by `$template` followed by `footer.php`, effectively outputting all three.

In fact, navigate your way back to `~/vhosts/pset7/templates` and open up `header.php` and `footer.php` in `gedit`. Ah, even more HTML! Thanks to render, those files' contents will be included at the top and bottom, respectively, of each of your pages. As a result, each of your pages will have access to Twitter's Bootstrap library, per the

link and script tags therein. And each page will have at least four `div` elements, three of which have unique IDs (`top`, `middle`, and `bottom`), if only to make styling them with CSS easier. Even more interestingly, though, notice how `header.php` conditionally outputs `$title`, if it is set. Remember how `index.php` contained the below line of code?

```
render("portfolio.php", ["title" => "Portfolio"]);
```

Well, because `render` calls `extract` on that second argument, an array, before requiring `header.php`, `header.php` ends up having access to a variable called `$title`. Neat, eh? You can pass even more values into a template simply by separating such key/value pairs with a comma, as in the below.

```
render("portfolio.php", ["cash" => 10000.00, "title" => "Portfolio"]);
```

Okay, now open up `constants.php` in `~/vhosts/pset7/includes` (which, recall, `config.php` required). Suffice it to say, this file defines a bunch of constants, but you shouldn't need to change any of them.

Navigate your way back to `~/vhosts/pset7/public` and open up `login.php`, another controller, with `gedit`. This controller's a bit more involved than `index.php` as it handles the authentication of users. Read through its lines carefully, taking note of how it how it queries the appliance's MySQL database using that `query` function from `functions.php`. That function (which we wrote) essentially simplifies use of PDO (PHP Data Objects), a library with which you can query MySQL (and other) databases. Per its definition in `functions.php`, the function accepts one or more arguments: a string of SQL followed by a comma-separated list of zero or more parameters that can be plugged into that string, not unlike `printf`. Whereas `printf` uses `%d`, `%s`, and the like for placeholders, though, `query` simply relies on question marks, no matter the type of value. And so the effect of

```
query("SELECT * FROM users WHERE username = ?", $_POST["username"]);
```

in `login.php` is to replace `?` with whatever username has been submitted (via POST) via an HTML form. (The function also ensures that any such placeholders' values are properly escaped so that your code is not vulnerable to "SQL injection attacks.") For instance, suppose that President Skroob tries to log into C$50 Finance by inputting his username and password. That line of code will ultimately execute the SQL statement below.

```
SELECT * FROM users WHERE username='skroob'
```

Beware, though. PHP is weakly (i.e., loosely) typed, and so functions like query can actually return different types. Indeed, even though query usually returns an array of rows (thanks to its invocation of PDO's `fetchAll`), it can also return `false` in case of errors. But, unlike `SELECT`s, some SQL queries (e.g., `DELETE`s, `UPDATE`s, and `INSERT`s) don't actually return rows, and so the array that `query` returns might sometimes be empty. When checking the return value of `query` for `false`, then, take care not to use `==`, because it turns out than an empty array is `==` to `false` because of implicit casting. But an empty array does not necessarily signify an error, only `false` does! Use, then, PHP's `===` (or `!==`) operator when checking return values for `false`, which compares its operands' values and types (not just their values), as in the below.

```
$result = query("INSERT INTO users (username, hash, cash) VALUES(?, ?, 10000.00)", $_POST["username"], crypt($_POST["password"]));
if ($result === false)
{
    // the INSERT failed, presumably because username already existed
}
```

See http://php.net/manual/en/language.operators.comparison.php for more details.

Anyhow, notice too that `login.php` "remembers" that a user is logged in by storing his or her unique ID inside of `$_SESSION`. As before, this controller does not contain any HTML. Rather, it calls `apologize` or renders `login_form.php` as needed. In fact, open up `login_form.php` in `~/vhosts/pset7/templates` with `gedit`. Most of that file is HTML that's stylized via some of Bootstrap's CSS classes, but notice how the HTML form therein POSTs to `login.php`. Just for good measure, take a peek at `apology.php` while you're in that directory as well. And also take a peek at `logout.php` back in `~/vhosts/pset7/public` to see how it logs out a user.

Alright, now navigate your way to `~/vhosts/pset7/public/css` and open up `styles.css` with `gedit`. Notice how this file already has a few "selectors" so that you don't have to include style attributes the elements matched by those selectors. No need to master CSS for this problem set, but do know that you should not have more than one `div` element per page whose `id` attribute has a value of `top`, more than one `div` element per page whose `id` attribute has a value of `middle`, or more than one `div` element per page whose `id` attribute has a value of `bottom`; an `id` must be unique. In any case, you are welcome to modify `styles.css` as you see fit.

You're also welcome to poke around `~/vhosts/pset7/public/js`, which contains some JavaScript files. But no need to use or write any JavaScript for this problem set. Those files are just there in case you'd like to experiment.

Phew, that was a lot. Help yourself to a snack.

- Alright, let's talk about that database we keep mentioning. So that you have someplace to store users' portfolios, the appliance comes with a MySQL database (called `pset7`). We've even pre-populated it with a table called `users` (which is why you were able to log in as President Skroob). Let's take a look.

  Head back to http://pset7/phpMyAdmin/ using Chrome inside of the appliance to access phpMyAdmin. Log in as John Harvard if prompted (with a username of **jharvard** and a password of **crimson**). You should then find yourself at phpMyAdmin's main page, in the top-left corner of which is that table called **users**. Click the name of that table to see its contents. Ah, some familiar folks. In fact, there's President Skroob's username and a hash of his password (which is the same as the combination to his luggage)!

  Now click the tab labeled **Structure**. Ah, some familiar fields. Recall that `login.php` generates queries like the below.

```
SELECT id FROM users WHERE username='skroob'
```

  As phpMyAdmin makes clear, this table called users contains three fields: `id` (the type of which is an `INT` that's `UNSIGNED`) along with `username` and `hash` (each of whose types is `VARCHAR`). It appears that none of these fields is allowed to be `NULL`, and the maximum length for each of each of `username` and `hash` is `255`. A neat feature of `id`, meanwhile, is that it will `AUTO_INCREMENT`: when inserting a new user into the table, you needn't specify a value for `id`; the user will be assigned the next available `INT`. Finally, if you click **Indexes**(above **Information**), you'll see that this table's `PRIMARY` key is `id`, the implication of which is that (as expected) no two users can share the same user ID. Recall that a primary key is a field with no duplicates (i.e., that is guaranteed to identify rows uniquely). Of course, `username` should also be unique across users, and so we have also defined it to be so (per the additional **Yes** under **Unique**). To be sure, we could have defined username as this table's primary key. But, for efficiency's sake, the more conventional approach is to use an `INT` like `id`. Incidentally, these fields are called "indexes" because, for primary keys and otherwise unique fields, databases tend to build "indexes," data structures that enable them to find rows quickly by way of those fields.

  Make sense?

- Okay, let's give each of your users some cash. Assuming you're still on phpMyAdmin's **Structure** tab, you should see a form with which you can add new columns. Click the radio button immediately to the left of **After**, select **hash** from the drop-down menu, as in the below, then click **Go**.

Via the form that appears, define a field called cash of type `DECIMAL` with a length of `65,4`, with a default value of `0.0000`, and with an attribute of `UNSIGNED`, as in the below, then click **Save**.

If you pull up the documentation for MySQL at http://dev.mysql.com/doc/refman/5.6/en/numeric-types.html, you'll see that the `DECIMAL` data type is used to "store exact numeric data values." A length of `65,4` for a `DECIMAL` means that values for `cash` can have no more than 65 digits in total, 4 of which can be to the right of the decimal point. (Ooo, fractions of pennies. Sounds like **Office Space**.)

Okay, return to the tab labeled **Browse** and give everyone $10,000.00 manually. (In theory, we could have defined `cash` as having a default value of `10000.000`, but, in general, best to put such settings in code, not your database, so that they're easier to change.) The easiest way is to click **Check All**, then click **Change** to the right of the pencil icon. On the page that appears, change `0.0000` to `10000.0000` for each of your users, then click **Go**. Won't they be happy!

- It's now time to code! Let's empower new users to register.

  Return to a terminal window, navigate your way to `~/vhosts/pset7/templates` and execute the below. (You are welcome, particularly if among those more comfortable, to stray from these filename conventions and structure your site as you see fit, so long as your implementation adheres to all other requirements.)

```
cp login_form.php register_form.php
```

  Then open up `register_form.php` with `gedit` and change the value of form's `action` attribute from `login.php` to `register.php`. Next add an additional field of type `password` to the HTML form called `confirmation` so that users are prompted to input their choice of passwords twice (to discourage mistakes). Finally, change the button's text from `Log In` to `Register` and change

```
or <a href="register.php">register</a> for an account
```

  to

```
or <a href="login.php">log in</a>
```

  so that users can navigate away from this page if they already have accounts.

Then, using `gedit`, create a new file called `register.php` with the contents below, taking care to save it in `~/vhosts/pset7/public`.

```
<?php

    // configuration
    require("../includes/config.php");

    // if form was submitted
    if ($_SERVER["REQUEST_METHOD"] == "POST")
    {
        // TODO
    }
    else
    {
        // else render form
        render("register_form.php", ["title" => "Register"]);
    }

?>
```

Alright, let's take a look at your work! Bring up http://pset7/login.php in Chrome inside of the appliance and click that page's link to `register.php`. You should then find yourself at http://pset7/register.php. If anything appears awry, feel free to make tweaks to `register_form.php` or `register.php`. Just be sure to save your changes and then reload the page in the browser.

Of course, `register.php` doesn't actually register users yet, so it's time to tackle that `TODO`! Allow us to offer some hints.

- If `$_POST["username"]` or `$_POST["password"]` is empty or if `$_POST["password"]` does not equal `$_POST["confirmation"]`, you'll want to inform registrants of their error.
- To insert a new user into your database, you might want to call

```
query("INSERT INTO users (username, hash, cash) VALUES(?, ?, 10000.00)", $_POST["username"], crypt($_POST["password"]));
```

though we leave it to you to decide how much cash your code should give to new users.

- Know that `query` will return `false` if your `INSERT` fails (as can happen if, say, `username` already exists). Be sure to check for false with `===` and not `==`.
- If, though, your `INSERT` succeeds, know that you can find out which `id` was assigned to that user with code like the below.

```
$rows = query("SELECT LAST_INSERT_ID() AS id");
```

```
$id = $rows[0]["id"];
```

- If registration succeeds, you might as well log the new user in (as by "remembering" that `id` in `$_SESSION`), thereafter redirecting to `index.php`.

Here's Zamyla with some additional hints:

- All done with `register.php`? Ready to test? Head back to http://pset7/register.php using Chrome inside of the appliance and try to register a new username. If you reach `index.php`, odds are you done good! Confirm as much by returning to phpMyAdmin, clicking once more that tab labeled **Browse** for the table called `users`. May that you see your new user. If not, it's time to debug!

  Be sure, incidentally, that any HTML generated by `register.php` is valid, as by ctrl- or right-clicking on the page in Chrome, selecting**View Page Source**, highlighting and copying the source code, and then pasting it into the W3C's validator athttp://validator.w3.org/#validate_by_input and then clicking **Check**. Ultimately, the **Result** of checking your page for validity via the W3C's validator should be **Passed** or **Tentatively passed**, in which case you should see a friendly green banner. Warnings are okay. Errors (and big red banners) are not. Note that you won't be able to "validate by URI" at http://validator.w3.org/#validate_by_uri, since your appliance isn't accessible on the public Internet!

- Do bear in mind as you proceed further that you are welcome to play with and learn from the staff's implementation of C$50 Finance athttps://www.cs50.net/finance.

  In particular, you are welcome to register with as many (fake) usernames as you would like in order to play. And you are welcome to view our pages' HTML and CSS (by viewing our source using your browser) so that you might learn from or improve upon our own design. If you wish, feel free to adopt our HTML and CSS as your own.

But do not feel that you need copy our design. In fact, for this problem set, you may modify every one of the files we have given you to suit your own tastes as well as incorporate your own images and more. In fact, may that your version of C$50 Finance be nicer than ours!

- Okay, now it's time to empower users to look up quotes for individual stocks. Odds are you'll want to create a new controller called, say, `quote.php` plus two new templates, the first of which displays an HTML form via which a user can submit a stock's symbol, the second of which displays, minimally, a stock's latest price (if passed, via render, an appropriate value).

  How to look up a stock's latest price? Well, recall that function called `lookup` in `functions.php`. Odds are you'll want to call it with code like the below.

```
$stock = lookup($_POST["symbol"]);
```

  Assuming the value of `$_POST["symbol"]` is a valid symbol for an actual stock, lookup will return an associative array with three keys for that stock, namely its `symbol`, its `name`, and its `price`. Know that you can use PHP's `number_format` function (somehow!) to format price to at least two decimal places but no more than four decimal places. See http://php.net/manual/en/function.number-format.php for details.

  Of course, if the user submits an invalid symbol (for which lookup returns false), be sure to inform the user somehow. Be sure, too, that any HTML generated by your templates is valid, per the W3C's validator.

  Here's Zamyla again:

- And now it's time to do a bit of design. At present, your database has no way of keeping track of users' portfolios, only users themselves. (By "portfolio," we mean a collection of stocks (i.e., shares of companies) that some user owns.) It doesn't really make sense to add additional fields to users itself in order to keep track of the stocks owned by users (using, say, one field per company owned). After all, how many different stocks might a user own? Better to maintain that data in a new table altogether so that we do not impose limits on users' portfolios or waste space with potentially unused fields.

  Exactly what sort of information need we keep in this new table in order to "remember" users' portfolios? Well, we probably want a field for users' IDs (`id`) so that we can cross-reference holdings with entries in `users`. We probably want to keep track of stocks owned by way of their symbols since those symbols are likely shorter (and thus more efficiently stored) than stocks' actual names. Of course, you could also assign unique numeric IDs to stocks and remember those instead of their symbols. But then you'd have to maintain your own database of companies, built up over time based on data from, say, Yahoo. It's probably better (and it's certainly simpler), then, to keep track of stocks simply by way of their symbols. And we probably want to keep track of how many shares a user owns of a particular stock. In other words, a table with three fields (`id`, `symbol`, and `shares`) sounds pretty good, but you're welcome to proceed with a design of your own. Whatever your decision, head back to phpMyAdmin and create this new table, naming it however you see fit. To

create a new table, click**pset7** in phpMyAdmin's top-left corner, and on the screen that appears, input a name for your table and some number of columns below**Create table**, then click **Go**. On the screen that appears next, define (in any order) each of your fields.

If you decide to go with three fields (namely `id`, `symbol`, and `shares`), realize that `id` should not be defined as a primary key in this table, else each user could own no more than one company's stock (since his or her `id` could not appear in more than one row). Realize, too, that you shouldn't let some `id` and some `symbol` to appear together in more than one row. Better to consolidate users' holdings by updating shares whenever some user sells or buys more shares of some stock he or she already owns. A neat way to impose this restriction while creating your table is to define a "joint primary key" by selecting an **Index** of `PRIMARY` for both `id` and `symbol`. That way,`INSERT` will fail if you try to insert more than one row for some pair of id and symbol. We leave it to you, though, to decide your fields' types. (If you include `id` in this table, know that its type should match that in `users`. But don't specify `AUTO_INCREMENT` for that field in this new table, as you only want auto-incrementation when user IDs are created for new users. And don't call your table `tbl`.) When done defining your table, click **Save**!

- Before we let users buy and sell stocks themselves, let's give some shares to President Skroob and friends at no charge. Click, in phpMyAdmin's left-hand frame, the link to `users` and remind yourself of your current users' IDs. Then click, in phpMyAdmin's left-hand frame, the link to your new table (for users' portfolios), followed by the tab labeled **Insert**. Via this interface, go ahead and "buy" some shares of some stocks on behalf of your users by manually inserting rows into this table. (You may want to return to Yahoo! Finance to look up some actual symbols.) No need to debit their `cash` in `users`; consider these shares freebies.

  Once you've bought your users some shares, let's see what you did. Click the tab labeled **SQL** and run a query like the below, where `tbl`represents your new table's name.

```
SELECT * FROM tbl WHERE id = 7
```

Assuming `7` is President Skroob's user ID, that query should return all rows from `tbl` that represent the president's holdings. If the only fields in table are, say, `id`, `symbol`, and `shares`, then know that the above is actually equivalent to the below.

```
SELECT id, symbol, shares FROM tbl WHERE id = 7
```

If, meanwhile, you'd like to retrieve only President Skroob's shares of Discovery Ventures, you might like to try a query like the below.

```
SELECT shares FROM tbl WHERE id = 7 AND symbol = 'DVN.V'
```

If you happened to buy President Skroob some shares of that company, the above should return one row with one column, the number of shares. If you did not get buy any such shares, the above will return an empty result set.

Incidentally, via this **SQL** tab, you could have inserted those "purchases" with `INSERT` statements. But phpMyAdmin's GUI saved you the trouble.

Alright, let's put this knowledge to use. It's time to let users peruse their portfolios! Overhaul `index.php` (a controller) and `portfolio.php` (a template) in such a way that they report each of the stocks in a user's portfolio, including number of shares and current price thereof, along with a user's current cash balance. Needless to say, `index.php` will need to invoke `lookup` much like `quote.php` did, though perhaps multiple times. And know that a PHP script can certainly invoke `query` multiple times, even though, thus far, we've seen it used in a file no more than once. And you can certainly iterate over the array it returns in a template (assuming you pass it in via `render`). For instance, if your goal is simply to display, say, President Skroob's holdings, one per row in some HTML table, you can generate rows with code like the below, where `$positions` is an array of associative arrays, each of which represents a position (i.e., a stock owned).

```
<table>
    <?php

        foreach ($positions as $position)
        {
            print("<tr>");
            print("<td>" . $position["symbol"] . "</td>");
            print("<td>" . $position["shares"] . "</td>");
            print("<td>" . $position["price"] . "</td>");
            print("</tr>");
        }

    ?>
</table>
```

Alternatively, you can avoid using the concatenation operator (`.`) via syntax like the below:

```
<table>
```

```
    <?php

        foreach ($positions as $position)
        {
            print("<tr>");
            print("<td>{$position["symbol"]}</td>");
            print("<td>{$position["shares"]}</td>");
            print("<td>{$position["price"]}</td>");
            print("</tr>");
        }

    ?>
</table>
```

Note that, in the above version, we've surrounded the lines of HTML with double quotes instead of single quotes so that the variables within (`$position["symbol"]`, `$position["shares"]`, and `$position["price"]`) are interpolated (i.e., substituted with their values) by PHP's interpreter; variables between single quotes are not interpolated. And we've also surrounded those same variables with curly braces so that PHP realizes they're variables; variables with simpler syntax (e.g., `$foo`) do not require the curly braces for interpolation. (It's fine to use double quotes inside those curly braces, even though we've also used double quotes to surround the entire argument to `print`.) Anyhow, though commonly done, generating HTML via calls to `print` isn't terribly elegant. An alternative approach, though still a bit inelegant, is code more like the below.

```
<?php foreach ($positions as $position): ?>

    <tr>
        <td><?= $position["symbol"] ?></td>
        <td><?= $position["shares"] ?></td>
        <td><?= $position["price"] ?></td>
    </tr>
```

```
<?php endforeach ?>
```

Of course, before you can even pass `$positions` to `portfolio.php`, you'll need to define it in `index.php`. Allow us to suggest code like the below, which combines names and prices from `lookup` with shares and symbols, as might be returned as `$rows` from `query`.

```
$positions = [];
foreach ($rows as $row)
{
    $stock = lookup($row["symbol"]);
    if ($stock !== false)
    {
        $positions[] = [
            "name" => $stock["name"],
            "price" => $stock["price"],
            "shares" => $row["shares"],
            "symbol" => $row["symbol"]
        ];
    }
}
```

Note that, with this code, we're deliberately create a new array of associative arrays (`$positions`) rather than add names and prices to an existing array of associative arrays (`$rows`). In the interests of good design, it's generally best not to alter functions' return values (like `$rows` from `query`).

Now, much like you can pass a page's title to render, so can you pass these positions, as with the below.

```
render("portfolio.php", ["positions" => $positions, "title" => "Portfolio"]);
```

Of course, you'll also need to pass a user's current cash balance from `index.php` to `portfolio.php` via `render` as well, but we leave it to you to figure out how.

To be clear, in the spirit of MVC, though, do take care not to call `lookup` inside of that (or any other) template; you should only call `lookup` in controllers. Even though templates (aka views) can contain PHP code, that code should only be used to print and/or iterate over data that's been passed in (as via render) from a controller.

As for what HTML to generate, look, as before, to https://www.cs50.net/finance/ for inspiration or hints. But do not feel obliged to mimic our design. Make this website your own! Although any HTML and PHP code that you yourself write should be pretty-printed (i.e., nicely indented), it's okay if lines exceed 80 characters in length. HTML that you generate dynamically (as via calls to `print`), though, does not need to be pretty-printed.

As before, be sure to display stocks' prices and users' cash balances to at least two decimal places but no more than four.

Incidentally, though we keep using President Skroob in examples, your code should work for whichever user is logged in.

As always, be sure that the HTML generated by `index.php` is valid.

Here's Zamyla with some additional tips:

- And now it is time to implement the ability to sell with a controller called, say, `sell.php` and some number of templates. We leave the design of this feature to you. But know that you can delete rows from your table (on behalf of, say, President Skroob) with SQL like the below.

```
DELETE FROM tbl WHERE id = 7 AND symbol = 'DVN.V'
```

We leave it to you to infer exactly what that statement should do. Of course, you could try the above out via phpMyAdmin's **SQL** tab. Now what about the user's cash balance? Odds are, your user is going to want the proceeds of all sales. So selling a stock involves updating not only your table for users' portfolios but `users` as well. We leave it to you to determine how to compute how much cash a user is owed upon sale of some stock. But once you know that amount (say, $500), SQL like the below should take care of the deposit (for, say, President Skroob).

```
UPDATE users SET cash = cash + 500 WHERE id = 7
```

Of course, if the database or web server happens to die between this `DELETE` and `UPDATE`, President Skroob might lose out on all of that cash. You need not worry about such cases! It's also possible, because of multithreading and, thus, race conditions, that a clever president could trick your site into paying out more than once. You need not worry about such cases either! Though, if you're so very inclined, you can employ SQL transactions (with InnoDB tables). See http://dev.mysql.com/doc/refman/5.6/en/innodb.html for reference.

It's fine, for simplicity, to require that users sell all shares of some stock or none, rather than only a few. Needless to say, try out your code by logging in as some user and selling some stuff. You can always "buy" it back manually with phpMyAdmin.

As always, be sure that your HTML is valid!

And as always, here is Zamyla!

- Now it's time to support actual buys. Implement the ability to buy, with a controller called, say, `buy.php` and some number of templates. (As before, you need not worry about interruptions of service or race conditions.) The interface with which you provide a user is entirely up to you, though, as before, feel free to look to https://www.cs50.net/finance for inspiration or hints. Of course, you'll need to ensure that a user cannot spend more cash than he or she has on hand. And you'll want to make sure that users can only buy whole shares of stocks, not fractions thereof. For this latter requirement, know that a call like

```
preg_match("/^\d+$/", $_POST["shares"])
```

will return `true` if and only if `$_POST["shares"]` contains a non-negative integer, thanks to its use of a regular expression. Seehttp://www.php.net/preg_match for details. Take care to apologize to the user if you must reject their input for any reason. In other words, be sure to perform rigorous error-checking. (We leave to you to determine what needs to be checked!)

When it comes time to store stocks' symbols in your database table, take care to store them in uppercase (as is convention), no matter how they were inputted by users, so that you don't accidentally treat, say, `dvn.v` and `DVN.V` as different stocks. Don't force users, though, to input symbols in uppercase.

Incidentally, if you implemented your table for users' portfolios as we did ours (with that joint primary key), know that SQL like the below (which, unfortunately, wraps onto two lines) will insert a new row into table unless the specified pair of `id` and `symbol` already exists in some row, in which case that row's number of shares will simply be increased (say, by `10`).

```
INSERT INTO table (id, symbol, shares) VALUES(7, 'DVN.V', 10) ON
DUPLICATE KEY UPDATE shares = shares + VALUES(shares)
```

As always, be sure to bang on your code. And be sure that your HTML is valid!

Here's Zamyla with some additional help:

- Alright, so your users can now buy and sell stocks and even check their portfolio's value. But they have no way of viewing their history of transactions.

  Enhance your implementations for buying and selling in such a way that you start logging transactions, recording for each:

  - Whether a stock was bought or sold.

    - The symbol bought or sold.
    - The number of shares bought or sold.
    - The price of a share at the time of transaction.
    - The date and time of the transaction.

    Then, by way of a controller called, say, `history.php` and some number of templates, enable users to peruse their own history of transactions, formatted as you see fit. Be sure that your HTML is valid!

    Here's Zamyla again:

- Phew. Glance back at `index.php` now and, if not there already, make that it somehow links to, at least, `buy.php`, `history.php`, `logout.php`, `quote.php`, and `sell.php` (or their equivalents) so that each is only one click away from a user's portfolio!
- And now the icing on the cake. Only one feature to go, but you get to choose. Implement at least one (1) of the features below. You may interpret each of the below as you see fit; we leave all design decisions to you. Be sure that your HTML is valid.
    - Empower users (who're already logged in) to change their passwords.
    - Empower users who've forgotten their password to reset it (as by having them register with an email address so that you can email them a link via which to do so).
    - Email users "receipts" anytime they buy or sell stocks.
    - Empower users to deposit additional funds.

    For tips on how to send email programmatically, see https://manual.cs50.net/Sending_Mail.

    Here's Zamyla with a few final thoughts:

Sanity Checks

Before you consider this problem set done, best to ask yourself these questions and then go back and improve your code as needed! Do not consider the below an exhaustive list of expectations, though, just some helpful reminders. The checkboxes that have come before these represent the exhaustive list! To be clear, consider the questions below rhetorical. No need to answer them in writing for us, since all of your answers should be "yes!"

- Is the HTML generated by all of your PHP files valid according to http://validator.w3.org/?
- Do your pages detect and handle invalid inputs properly?
- Are you recording users' histories of transactions properly?
- Did you add one (1) additional feature of your own?
- Did you choose appropriate data types for your database tables' fields?

- Are you displaying any dollar amounts to at least two decimal places but no more than four?
- Are you storing stocks' symbols in your table(s) in uppercase?

As always, if you can't answer "yes" to one or more of the above because you're having some trouble, do turn to cs50.net/discuss!

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and "export" your MySQL database (i.e., save it into a text file) by executing the commands below, inputting **crimson** when prompted for a password. For security, you won't see the password as you type it.

```
cd ~/vhosts/pset7
```

```
mysqldump -u jharvard -p pset7 > pset7.sql
```

If you type `ls` thereafter, you should see that you have a new file called `pset7.sql` in `~/vhosts/pset7`. (If you realize later that you need to make a change to your database and re-export it, you can delete `pset7.sql` with `rm pset7.sql`, then re-export as before.) Next create a ZIP (i.e., compressed) file containing your entire `pset7` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset7`, including any subdirectories (or even subsubdirectories!).

```
cd ~/vhosts
zip -r pset7.zip pset7/
```

If you type `ls` thereafter, you should see that you have a new file called `pset7.zip` in `~/vhosts`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset7.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit CS50 Submit, logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 7** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.

- Navigate your way to `pset7.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset7.zip`, click it once to select it, then click **Open**.
- Click **Start upload** to upload your ZIP file to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to CS50 Submit and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://x.cs50.net/2014/psets/7/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 7.

Problem Sets 1 -5:
Hacker Edition

Problem Set 1: C

This is the Hacker Edition of Problem Set 1. It cannot be submitted for credit.

Objectives

- Get comfortable with Linux.
- Start thinking more carefully.
- Solve some problems in C.

Recommended Reading

- Pages 1 – 7, 9, and 10 of http://www.howstuffworks.com/c.htm.
- Chapters 1 – 6 of *Programming in C*.

diff pset1 hacker1

- Hacker Edition plays with credit cards instead of coins.
- Hacker Edition demands two half pyramids.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.

- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Started

- Recall that the CS50 Appliance is a "virtual machine" (running an operating system called Fedora, which itself is a flavor of Linux) that you can run inside of a window on your own computer, whether you run Windows, Mac OS, or even Linux itself. To do so, all you need is a "hypervisor" (otherwise known as a "virtual machine monitor"), software that tricks the appliance into thinking that it's running on "bare metal."

  Alternatively, you could buy a new computer, install Fedora on it (i.e., bare metal), and use that! But a hypervisor lets you do all that for free with whatever computer you already have. Plus, the CS50 Appliance is pre-configured for CS50, so, as soon as you install it, you can hit the ground running.

  So let's get a hypervisor and the CS50 Appliance installed on your computer. Head tohttps://manual.cs50.net/appliance/19/#how_to_install_appliance, where instructions await. In particular, if running Mac OS, follow the instructions for VMware Fusion. If running Windows or Linux, follow the instructions for VMware Player.

- Once you have the CS50 Appliance installed, go ahead and start it (per those same instructions). A small window should open, inside of which the appliance should boot. A few seconds or minutes later, you should find yourself logged in as John Harvard (whose username is**jharvard** and whose password is **crimson**), with John Harvard's desktop before you.

  If you find that the appliance runs unbearably slow on your PC, particularly if several years old or a somewhat slow netbook, or if you see a hint about "long mode," try the instructions at https://manual.cs50.net/virtualization.

  Feel free to poke around, particularly the 50 Menu in the appliance's bottom-left corner. You should find the graphical user interface (GUI), called Xfce, reminiscent of both Mac OS and Windows. Linux actually comes with a bunch of GUIs; Xfce is just one. If you're already familiar with Linux, you're welcome to install other software via **Menu > Administration > Add/Remove Software**, but the appliance should have everything you need for now. You're also welcome to play with the appliance's various features, per the instructions athttps://manual.cs50.net/appliance/19/#how_to_use_appliance, but this problem set will explicitly mention anything that you need know or do.

- Even if you just downloaded the appliance, ensure that it's completely up-to-date by opening a terminal window, as via **Menu > Programming > Terminal**, typing

```
update50
```

and then hitting Enter on your keyboard. So long as your computer (and, thus, the appliance) has Internet access, the appliance should proceed to download and install any available updates.

- Next, follow the instructions at https://manual.cs50.net/appliance/19/#how_to_synchronize_files_with_dropbox to configure the appliance to use Dropbox so that your work is automatically backed up, just in case something goes wrong with your appliance. (If you really don't want to use Dropbox, that's fine, but realize your files won't be backed up as a result!) If you don't yet have a Dropbox account, sign up when prompted for the free (2 GB) plan. You're welcome to install Dropbox on your own computer as well (outside of the appliance), perhttps://www.dropbox.com/install, but no need if you'd rather not; just inside the appliance is fine.

  If you're already a Dropbox user but don't want your personal files to be synched into the appliance, simply enable **Selective Sync**, per the CS50 Manual's instructions.

- Okay, let's create a folder (otherwise known as a "directory") in which your code for this problem set will soon live. Go ahead and double-click **Home** on John Harvard's desktop (in the appliance's top-left corner). A window entitled **Home** should appear, indicating that you're inside of John Harvard's "home directory" (i.e., personal folder). Then double-click the folder called **Dropbox**, at which point the window's title should change to **Dropbox**. Next select **New Folder** under the gear icon in the window's top-right corner, at which point a new folder called **Untitled Folder** should appear. Rename it **hacker1** (in all lowercase, with no spaces). (If the folder's name doesn't seem to be editable, control-click (i.e., click while holding your keyboard's control key) the **Untitled Folder** once, then select **Rename…**, at which point its name should become editable.) Then double-click that **hacker1** folder to open it. The window's title should change to **hacker1**, and you should see an otherwise empty folder (since you just created it). Notice, though, that atop the window are three buttons, **Home**,**Dropbox**, and **hacker1**, that indicate where you were and where you are; you can click buttons like those to navigate back and forth easily.

- Okay, go ahead and close any open windows, then select **Menu > Programming > gedit**. (Recall that Menu is in the appliance's bottom-left corner.) A window entitled **Unsaved Document 1 - gedit** should appear, inside of which is a tab entitled **Unsaved Document 1**. Clearly the document is just begging to be saved. Go ahead and type `hello` (or the ever-popular `asdf`) in the tab, and then notice how the tab's name is now prefixed with an asterisk (*), indicating that you've made changes since the file was first opened. Select **File > Save**, and a window entitled **Save As** should appear. Input `hello.txt` next to **Name**, then click **jharvard** under **Places**. You should then see the contents of John Harvard's home directory. Double-click **Dropbox**, then double-click **hacker1**, and you should find yourself inside that empty folder you created. Now, at the bottom of this same window, you should see that the file's default **Character Encoding** is**Unicode (UTF-8)** and that the file's default **Line Ending** is **Unix/Linux**. No need to change either; just notice they're there. That the file's**Line Ending** is **Unix/Linux** just means that `gedit` will insert (invisibly) `\n` at the end of any line of text that you type. Windows, by contrast, uses `\r\n`, and Mac OS uses `\r`, but more on those details some other time.

- Okay, click **Save** in the window's bottom-right corner. The window should close, and you should see that the original window's title is now**hello.txt (~/Dropbox/hacker1) - gedit**. The parenthetical just means that **hello.txt** is inside of **hacker1**, which is inside of **Dropbox**, which is inside of ~, which is shorthand notation for John Harvard's home directory. A useful reminder is all. The tab, meanwhile, should now be entitled **hello.txt** (with no asterisk, unless you accidentally hit the keyboard again).
- Okay, with `hello.txt` still open in `gedit`, notice that beneath your document is a "terminal window," a command-line (i.e., text-based) interface via which you can navigate the appliance's hard drive and run programs (by typing their name). Notice that the window's "prompt" is

```
jharvard@appliance (~):
```

which means that you are logged into the appliance as John Harvard and that you are currently inside of ~ (i.e., John Harvard's home directory). If that's the case, there should be a **Dropbox** directory somewhere inside. Let's confirm as much.

Click somewhere inside of that terminal window, and the prompt should start to blink. Type

```
ls
```

and then Enter. That's a lowercase L and a lowercase S, which is shorthand notation for "list." Indeed, you should then see a list of the folders inside of John Harvard's home directory, among which is **Dropbox**! Let's open that folder, followed immediately by the **hacker1**folder therein. Type

```
cd Dropbox/hacker1
```

or even

```
cd ~/Dropbox/hacker1
```

followed by Enter to change your directory to **~/Dropbox/hacker1** (ergo, `cd`). You should find that your prompt changes to

```
jharvard@appliance (~/Dropbox/hacker1):
```

confirming that you are indeed now inside of **~/Dropbox/hacker1** (i.e., a directory called **hacker1** inside of a directory called **Dropbox**inside of John Harvard's home directory). Now type

```
ls
```

followed by Enter. You should see **hello.txt**! Now, you can't click or double-click on that file's name there; it's just text. But that listing does confirm that hello.txt is where we hoped it would be.

Let's poke around a bit more. Go ahead and type

```
cd
```

and then Enter. If you don't provide `cd` with a "command-line argument" (i.e., a directory's name), it whisks you back to your home directory by default. Indeed, your prompt should now be:

```
jharvard@appliance (~):
```

Phew, home sweet home. Make sense? If not, no worries; it soon will! It's in this terminal window that you'll soon be compiling your first program! For now, though, close `gedit` (via **File > Quit**) and, with it, **hello.txt**.

- Incidentally, if the need arises, know that you can transfer files to and from the appliance per the instructions at https://manual.cs50.net/appliance/19/#how_transfer_files_between_appliance_and_your_computer.

hello, world

- Shall we have you write your first program?

  Okay, go ahead and launch `gedit`. (Remember how?) You should find yourself faced with another **Unsaved Document 1**. Go ahead and save the file as `hello.c` (not `hello.txt`) inside of `hacker1`, just as before. (Remember how?) Once the file is saved, the window's title should change to **hello.c (~/Dropbox/hacker1) - gedit**, and the tab's title should change to **hello.c**. (If either does not, best to close `gedit` and start fresh! Or ask for help!)

  Go ahead and write your first program by typing these lines into the file (though you're welcome to change the words between quotes to whatever you'd like):

```
#include <stdio.h>


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

Notice how `gedit` adds "syntax highlighting" (i.e., color) as you type. Those colors aren't actually saved inside of the file itself; they're just added by `gedit` to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, `gedit` wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent.

Do be sure that you type in this program just right, else you're about to experience your first bug! In particular, capitalization matters, so don't accidentally capitalize words (unless they're between those two quotes). And don't overlook that one semicolon. C is quite nitpicky!

When done typing, select **File > Save** (or hit ctrl-s), but don't quit. Recall that the leading asterisk in the tab's name should then disappear. Click anywhere in the terminal window beneath your code, and its prompt should start blinking. But odds are the prompt itself is just

```
jharvard@appliance (~):
```

which means that, so far as the terminal window's concerned, you're still inside of John Harvard's home directory, even though you saved the program you just wrote inside of `~/Dropbox/hacker1` (per the top of `gedit`'s window). No problem, go ahead and type

```
cd Dropbox/hacker1
```

or

```
cd ~/Dropbox/hacker1
```

at the prompt, and the prompt should change to

```
jharvard@appliance (~/Dropbox/hacker1):
```

in which case you're where you should be! Let's confirm that `hello.c` is there. Type

```
ls
```

at the prompt followed by Enter, and you should see both `hello.c` and `hello.txt`? If not, no worries; you probably just missed a small step. Best to restart these past several steps or ask for help!

Assuming you indeed see hello.c, let's try to compile! Cross your fingers and then type

```
make hello
```

at the prompt, followed by Enter. (Well, maybe don't cross your fingers whilst typing.) To be clear, type only `hello` here, not `hello.c`. If all that you see is another, identical prompt, that means it worked! Your source code has been translated to 0s and 1s that you can now execute. Type

```
./hello
```

at your prompt, followed by Enter, and you should see whatever message you wrote between quotes in your code! Indeed, if you type

```
ls
```

followed by Enter, you should see a new file, `hello`, alongside `hello.c` and `hello.txt`.

If, though, upon running `make`, you instead see some error(s), it's time to debug! (If the terminal window's too small to see everything, click and drag its top border upward to increase its height.) If you see an error like expected declaration or something no less mysterious, odds are you made a syntax error (i.e., typo) by omitting some character or adding something in the wrong place. Scour your code for any differences vis-à-vis the template above. It's easy to miss the slightest of things when learning to program, so do compare your code against ours character by character; odds are the mistake(s) will jump out! Anytime you make changes to your own code, just remember to re-save via **File > Save** (or ctrl-s), then re-click inside of the terminal window, and then re-type

```
make hello
```

at your prompt, followed by Enter. (Just be sure that you are inside of `~/Dropbox/hacker1` within your terminal window, as your prompt will confirm or deny.) If you see no more errors, try running your program by typing

```
./hello
```

at your prompt, followed by Enter! Hopefully you now see precisely the below?

```
hello, world
```

If not, reach out to CS50 Discuss for help!

Incidentally, if you find `gedit`'s built-in terminal window too small for your tastes, know that you can open one in its own window via **Menu > Programming > Terminal**. You can then alternate between `gedit` and `Terminal` as needed, as by clicking either's name along the appliance's bottom.

Woo hoo! You've begun to program!

This is CS50 Check.

- Now let's see if the program you just wrote is correct! Included in the CS50 Appliance is `check50`, a command-line program with which you can check the correctness of (some of) your programs.

  If not already there, navigate your way to `~/Dropbox/hacker1` by executing the command below.

```
cd ~/Dropbox/hacker1
```

  If you then execute

```
ls
```

  you should see, at least, `hello.c`. Be sure it's indeed spelled `hello.c` and not `Hello.c`, `hello.C`, or the like. If it's not, know that you can rename a file by executing

```
mv source destination
```

  where `source` is the file's current name, and `destination` is the file's new name. For instance, if you accidentally named your program `Hello.c`, you could fix it as follows.

```
mv Hello.c hello.c
```

  Okay, assuming your file's name is definitely spelled `hello.c` now, go ahead and execute the below. Note that `2013.hacker1.hello` is just a unique identifier for this problem's checks.

```
check50 2014/x/hacker1/hello hello.c
```

  Assuming your program is correct, you should then see output like

```
:) hello.c exists
:) hello.c compiles
:) prints "hello, world\n"
```

  where each green smiley means your program passed a check (i.e., test). You may also see a URL at the bottom of `check50`'s output, but that's just for staff (though you're welcome to visit it).

If you instead see yellow or red smileys, it means your code isn't correct! For instance, suppose you instead see the below.

```
:( hello.c exists
   \ expected hello.c to exist
:| hello.c compiles
   \ can't check until a frown turns upside down
:| prints "hello, world\n"
   \ can't check until a frown turns upside down
```

Because `check50` doesn't think `hello.c` exists, as per the red smiley, odds are you uploaded the wrong file or misnamed your file. The other smileys, meanwhile, are yellow because those checks are dependent on `hello.c` existing, and so they weren't even run.

Suppose instead you see the below.

```
:) hello.c exists
:) hello.c compiles
:( prints "hello, world\n"
   \ expected output, but not "hello, world"
```

Odds are, in this case, you printed something other than `hello, world\n` verbatim, per the spec's expectations. In particular, the above suggests you printed `hello, world`, without a trailing newline (`\n`).

Know that `check50` won't actually record your scores in CS50's gradebook. Rather, it lets you check your work's correctness *before* you submit your work. Once you actually submit your work (per the directions at this spec's end), CS50's staff will use `check50` to evaluate your work's correctness officially.

Shorts

- Head to Week 1's shorts and curl up with Nate's short on libraries.
  - What's a pre-processor? How does

    ```
    #include <cs50.h>
    ```

    relate?

    - What's a compiler?
    - What's an assembler?
    - What's a linker? How does

      ```
      -lcs50
      ```

      relate?
- Curl up with at least two other shorts from Week 1. Some additional questions may be in your future!

Bad Credit

- Odds are you have a credit card in your wallet. Though perhaps the bill does not (yet) get sent to you! That card has a number, both printed on its face and embedded (perhaps with some other data) in the magnetic stripe on back. That number is also stored in a database somewhere, so that when your card is used to buy something, the creditor knows whom to bill. There are a lot of people with credit cards in this world, so those numbers are pretty long: American Express uses 15-digit numbers, MasterCard uses 16-digit numbers, and Visa uses 13- and 16-digit numbers. And those are decimal numbers (0 through 9), not binary, which means, for instance, that American Express could print as many as 10^(15) = 1,000,000,000,000,000 unique cards! (That's, ahem, a quadrillion.)

  Now that's a bit of an exaggeration, because credit card numbers actually have some structure to them. American Express numbers all start with 34 or 37; MasterCard numbers all start with 51, 52, 53, 54, or 55; and Visa numbers all start with 4. But credit card numbers also have a "checksum" built into them, a mathematical relationship between at least one number and others. That checksum enables computers (or humans who like math) to detect typos (e.g., transpositions), if not fraudulent numbers, without having to query a database, which can be slow. (Consider the awkward silence you may have experienced at some point whilst paying by credit card at a store whose computer uses a dial-up modem to verify your card.) Of course, a dishonest mathematician could certainly craft a fake number that nonetheless respects the mathematical constraint, so a database lookup is still necessary for more rigorous checks.

  So what's the secret formula? Well, most cards use an algorithm invented by Hans Peter Luhn, a nice fellow from IBM. According to Luhn's algorithm, you can determine if a credit card number is (syntactically) valid as follows:

  1. Multiply every other digit by 2, starting with the number's second-to-last digit, and then add those products' digits together.
  2. Add the sum to the sum of the digits that weren't multiplied by 2.
  3. If the total's last digit is 0 (or, put more formally, if the total modulo 10 is congruent to 0), the number is valid!

  That's kind of confusing, so let's try an example with Nate's AmEx: 378282246310005.

4. For the sake of discussion, let's first underline every other digit, starting with the number's second-to-last digit:

   3782822463100̲5

   Okay, let's multiply each of the underlined digits by 2:

   7•2 + 2•2 + 2•2 + 4•2 + 3•2 + 0•2 + 0•2

   That gives us:

   14 + 4 + 4 + 8 + 6 + 0 + 0

   Now let's add those products' digits (i.e., not the products themselves) together:

   1 + 4 + 4 + 4 + 8 + 6 + 0 + 0 = 27

5. Now let's add that sum (27) to the sum of the digits that weren't multiplied by 2:

   27 + 3 + 8 + 8 + 2 + 6 + 1 + 0 + 5 = 60

6. Yup, the last digit in that sum (60) is a 0, so Nate's card is legit!

So, validating credit card numbers isn't hard, but it does get a bit tedious by hand. Let's write a program.

In `credit.c`, write a program that prompts the user for a credit card number and then reports (via `printf`) whether it is a valid American Express, MasterCard, or Visa card number, per the definitions of each's format herein. So that we can automate some tests of your code, we ask that your program's last line of output be `AMEX\n` or `MASTERCARD\n` or `VISA\n` or `INVALID\n`, nothing more, nothing less, and that `main` always return `0`. For simplicity, you may assume that the user's input will be entirely numeric (i.e., devoid of hyphens, as might be printed on an actual card). But do not assume that the user's input will fit in an `int`! Best to use `GetLongLong` from CS50's library to get users' input. (Why?)

Of course, to use `GetLongLong`, you'll need to tell `clang` about CS50's library. Be sure to put

```
#include <cs50.h>
```

toward the top of `credit.c`. And be sure to compile your code with a command like the below.

```
clang -o credit credit.c -lcs50
```

Note that `-lcs50` must come at this command's end because of how clang works.

Incidentally, recall that `make` can invoke `clang` for you and provide that flag for you, as via the command below.

```
make credit
```

Assuming your program compiled without errors (or, ideally, warnings) via either command, you can run your program with the command below.

```
./credit
```

Consider the below representative of how your own program should behave when passed a valid credit card number (sans hyphens); highlighted in bold is some user's input.

```
jharvard@appliance (~/Dropbox/hacker1): ./credit
Number: 378282246310005
AMEX
```

Of course, `GetLongLong` itself will reject hyphens (and more) anyway:

```
jharvard@appliance (~/Dropbox/hacker1): ./credit
Number: 3782-822-463-10005
Retry: foo
Retry: 378282246310005
AMEX
```

But it's up to you to catch inputs that are not credit card numbers (e.g., Lauren's phone number), even if numeric:

```
jharvard@appliance (~/Dropbox/hacker1): ./credit
Number: 7722574501
INVALID
```

Test out your program with a whole bunch of inputs, both valid and invalid. (We certainly will!) Here are a few card numbers that PayPal recommends for testing:

https://www.paypalobjects.com/en_US/vhelp/paypalmanager_help/credit_card_numbers.htm

Google (or perhaps a roommate's wallet) should turn up more. (If your roommate asks what you're doing, don't mention us.) If your program behaves incorrectly on some inputs (or doesn't compile at all), time to debug!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/hacker1/credit credit.c
```

And if you'd like to play with the staff's own implementation of `credit` in the appliance, you may execute the below **starting Tue 9/17**.

```
~cs50/hacker1/credit
```

Itsa Mario

- Toward the beginning of World 1-1 in Nintendo's Super Mario Brothers, Mario must hop over two "half-pyramids" of blocks as he heads toward a flag pole. Below is a screenshot.

Write, in a file called `mario.c` in your `~/Dropbox/hacker1` directory, a program that recreates these half-pyramids using hashes (`#`) for blocks. However, to make things more interesting, first prompt the user for the half-pyramids' heights, a non-negative integer no greater than `23`. (The height of the half-pyramids pictured above happens to be `4`, the width of each half-pyramid `4`, with an a gap of size `2` separating them.) Then, generate (with the help of `printf` and one or more loops) the desired half-pyramids. Take care to left-align the bottom-left corner of the left-hand half-pyramid, as in the sample output below, wherein boldfaced text represents some user's input.

```
jharvard@appliance (~/Dropbox/hacker1): ./mario
Height: 4
   #  #
  ##  ##
 ###  ###
####  ####
```

No need to generate the bricks, cloud, numbers, or text in the sky or Mario himself. Just the half-pyramids! And be sure that main returns `0`.

We leave it to you to determine how to compile and run this particular program!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/hacker1/mario mario.c
```

And if you'd like to play with the staff's own implementation of `mario` in the appliance, you may execute the below **starting Tue 9/17**.

```
~cs50/hacker1/mario
```

Problem Set 2: Crypto

This is the Hacker Edition of Problem Set 3. It cannot be submitted for credit.

Objectives

- Introduce you to larger programs and programs with multiple source files.
- Accustom you to reading someone else's code.
- Empower you with Makefiles.
- Introduce you to literature in computer science.
- Implement a party favor.

Recommended Reading

- Page 17 of http://www.howstuffworks.com/c.htm.
- Chapters 13, 15, and 18 of *Programming in C*.

diff pset3 hacker3

- Hacker Edition dares you to implement sort in $O(n)$ instead of $O(n^2)$.
- Hacker Edition asks you to play God.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.

- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.

- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 3's shorts and watch the shorts on bubble sort, insertion sort, and selection sort. Then head to Week 4's shorts and watch the short on `gdb`. (Phew, so many shorts! And so many sorts! Ha.)
  - `gdb` lets you "debug" program, but, more specifically, what does it let you do?
  - Why does binary search require that an array be sorted?
  - Why is bubble sort in $O(n^2)$?
  - Why is insertion sort in $\Omega(n)$?
  - What's the worst-case running time of merge sort?
  - In no more than 3 sentences, how does selection sort work?

Getting Started

- Recall that, for Problem Sets 1 and 2, you started writing programs from scratch, creating your own `pset1` and `pset2` directories with `mkdir`. For Problem Set 3, you'll instead download "distribution code" (otherwise known as a "distro"), written by us, and add your own lines of code to it. You'll first need to read and understand our code, though, so this problem set is as much about learning to read someone else's code as it is about writing your own!

  Let's get you started. Go ahead and open a terminal window if not open already (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

  ```
  update50
  ```

  to make sure your appliance is up-to-date. Then execute

  ```
  cd ~/Dropbox
  ```

  followed by

  ```
  wget http://cdn.cs50.net/2013/fall/psets/3/hacker3/hacker3.zip
  ```

  to download a ZIP of this problem set's distro into your appliance (with a command-line program called `wget`). You should see a bunch of output followed by:

```
'hacker3.zip' saved
```

If you instead see

```
unable to resolve host address
```

your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

Ultimately, confirm that you've indeed downloaded `hacker3.zip` by executing:

```
ls
```

Then, run

```
unzip hacker3.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `hacker3` as well. Proceed to execute

```
cd hacker3
```

followed by

```
ls
```

and you should see that the directory contains two "subdirectories":

```
fifteen  find
```

Fun times ahead!

Searching

- Okay, let's dive into the first of those subdirectories. Execute the command below in a terminal window in your appliance.

```
cd ~/hacker3/find/
```

If you list the contents of this directory, you should see the below.

```
helpers.c  helpers.h  Makefile  find.c  generate.c
```

Wow, that's a lot of files, eh? Not to worry, we'll walk you through them.

- Implemented in `generate.c` is a program that uses a "pseudorandom-number generator" (via a function called `rand`) to generate a whole bunch of random (well, pseudorandom, since computers can't actually generate truly random) numbers, one per line. (Cf.https://www.cs50.net/resources/cppreference.com/stdother/rand.html.) Go ahead and compile this program by executing the command below.

```
make generate
```

Now run the program you just compiled by executing the command below.

```
./generate
```

You should be informed of the program's proper usage, per the below.

```
Usage: generate n [s]
```

As this output suggests, this program expects one or two command-line arguments. The first, `n`, is required; it indicates how many pseudorandom numbers you'd like to generate. The second, `s`, is optional, as the brackets are meant to imply; if supplied, it represents the value that the pseudorandom-number generator should use as its "seed." A seed is simply an input to a pseudorandom-number generator that influences its outputs. For instance, if you seed `rand` by first calling `srand` (another function whose purpose is to "seed" `rand`) with an argument of, say, `1`, and then call `rand` itself three times, `rand` might return `17767`, then `9158`, then `39017`. (Cf.https://www.cs50.net/resources/cppreference.com/stdother/srand.html.) But if you instead seed `rand` by first calling `srand` with an argument of, say, `2`, and then call `rand` itself three times, `rand` might instead return `38906`, then `31103`, then `52464`. But if you re-seed `rand` by calling `srand` again with an argument of `1`, the next three times you call `rand`, you'll again get `17767`, then `9158`, then `39017`! See, not so random.

Go ahead and run this program again, this time with a value of, say, `10` for `n`, as in the below; you should see a list of 10 pseudorandom numbers.

```
./generate 10
```

Run the program a third time using that same value for n; you should see a different list of 10 numbers. Now try running the program with a value for s too (e.g., 0), as in the below.

```
./generate 10 0
```

Now run that same command again:

```
./generate 10 0
```

Bet you saw the same "random" sequence of ten numbers again? Yup, that's what happens if you don't vary a pseudorandom number generator's initial seed.

- Now take a look at `generate.c` itself with `gedit`. (Remember how?) Comments atop that file explain the program's overall functionality. But it looks like we forgot to comment the code itself. Read over the code carefully until you understand each line and then comment our code for us, replacing each `TODO` with a phrase that describes the purpose or functionality of the corresponding line(s) of code. (Know that an `unsigned int` is just an `int` that cannot be negative.) And for more details on `rand` and `srand`, recall that you can execute:

- `man rand`

```
man srand
```

Once done commenting `generate.c`, re-compile the program to be sure you didn't break anything by re-executing the command below.

```
make generate
```

If `generate` no longer compiles properly, take a moment to fix what you broke!

Now, recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. Notice, in fact, how `make` just executed a pretty long command for you, per the tool's output. However, as your programs grow in size, `make` won't be able to infer from context anymore how to compile your code; you'll need to start telling `make` how to compile your program, particularly when they involve multiple source (i.e., `.c`) files. And so we'll start relying on "Makefiles," configuration files that tell `make` exactly what to do.

How did `make` know how to compile generate in this case? It actually used a configuration file that we wrote. Using `gedit`, go ahead and look at the file called `Makefile` that's in the same directory as `generate.c`. This `Makefile` is essentially a list of rules that we wrote for you that tells make how to build generate from `generate.c` for you. The relevant lines appear below.

```
generate: generate.c
    `clang` -ggdb -std=c99 -Wall -Werror -o generate generate.c
```

The first line tells `make` that the "target" called `generate` should be built by invoking the second line's command. Moreover, that first line tells `make` that `generate` is dependent on `generate.c`, the implication of which is that `make` will only re-build generate on subsequent runs if that file was modified since `make` last built `generate`. Neat time-saving trick, eh? In fact, go ahead and execute the command below again, assuming you haven't modified `generate.c`.

```
make generate
```

You should be informed that generate is already up to date. Incidentally, know that the leading whitespace on that second line is not a sequence of spaces but, rather, a tab. Unfortunately, `make` requires that commands be preceded by tabs, so be careful not to change them to spaces with `gedit` (which automatically converts tabs to four spaces), else you may encounter strange errors! The `-Werror` flag, recall, tells `clang` to treat warnings (bad) as though they're errors (worse) so that you're forced (in a good, instructive way!) to fix them.

- Now take a look at `find.c` with `gedit`. Notice that this program expects a single command-line argument: a "needle" to search for in a "haystack" of values. Once done looking over the code, go ahead and compile the program by executing the command below.

```
make find
```

Notice, per that command's output, that `make` actually executed the below for you.

```
clang -ggdb -std=c99 -Wall -Werror -o find find.c helpers.c -lcs50 -
lm
```

Notice further that you just compiled a program comprising not one but two `.c` files: `helpers.c` and `find.c`. How did `make` know what to do? Well, again, open up `Makefile` to see the man behind the curtain. The relevant lines appear below.

```
find: find.c helpers.c helpers.h
    clang -ggdb -std=c99 -Wall -Werror -o find find.c helpers.c -
lcs50 -lm
```

Per the dependencies implied above (after the colon), any changes to `find.c`, `helpers.c`, or `helpers.h` will compel `make` to rebuild find the next time it's invoked for this target.

Go ahead and run this program by executing, say, the below.

```
./find 13
```

You'll be prompted to provide some hay (i.e., some integers), one "straw" at a time. As soon as you tire of providing integers, hit ctrl-d to send the program an `EOF` (end-of-file) character. That character will compel `GetInt` from the CS50 Library to return `INT_MAX`, a constant that, per `find.c`, will compel `find` to stop prompting for hay. The program will then look for that needle in the hay you provided, ultimately reporting whether the former was found in the latter. In short, this program searches an array for some value. At least, it should, but it won't find anything yet! That's where you come in. More on your role in a bit.

In turns out you can automate this process of providing hay, though, by "piping" the output of `generate` into `find` as input. For instance, the command below passes 1,000 pseudorandom numbers to `find`, which then searches those values for `42`.

```
./generate 1000 | ./find 42
```

Note that, when piping output from `generate` into `find` in this manner, you won't actually see `generate`'s numbers, but you will see `find`'s prompts.

Alternatively, you can "redirect" `generate`'s output to a file with a command like the below.

```
./generate 1000 > numbers.txt
```

You can then redirect that file's contents as input to `find` with the command below.

```
./find 42 < numbers.txt
```

Let's finish looking at that `Makefile`. Notice the line below.

```
all: find generate
```

This target implies that you can build both `generate` and `find` simply by executing the below.

```
make all
```

Even better, the below is equivalent (because `make` builds a `Makefile`'s first target by default).

```
make
```

If only you could whittle this whole problem set down to a single command! Finally, notice these last lines in `Makefile`:

```
clean:
    rm -f *.o a.out core find generate
```

This target allows you to delete all files ending in `.o` or called `core` (more on that soon!), `find`, or `generate` simply by executing the command below.

```
make clean
```

Be careful not to add, say, `*.c` to that last line in `Makefile`! (Why?) Any line, incidentally, that begins with `#` is just a comment.

- And now the fun begins! Notice that `find.c` calls `search`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully in `helpers.c`! (To be sure, we could have put the contents of `helpers.h` and `helpers.c` in `find.c` itself. But it's sometimes better to organize programs into multiple files, especially when some functions are essentially utility functions that might later prove useful to other programs as well, much like those in the CS50 Library.) Take a peek at `helpers.c` with `gedit`, and you'll see that `search` always returns `false`, whether or not `value` is in `values`. Re-write `search` in such a way that it uses linear search, returning `true` if `value` is in `values` and `false` if `value` is not in `values`. Take care to return `false` right away if `n` isn't even positive.

  When ready to check the correctness of your program, try running the command below.

```
./generate 1000 50 | ./find 2008
```

Because one of the numbers outputted by `generate`, when seeded with `50`, is `2008`, your code should find that "needle"! By contrast, try running the command below as well.

```
./generate 1000 50 | ./find 2013
```

Because `2013` is not among the numbers outputted by `generate`, when seeded with `50`, your code shouldn't find that needle. Best to try some other tests as well, as by running `generate` with some seed, taking a look at its output, then piping that same output to `find`, looking for a "needle" you know to be among the "hay".

Incidentally, note that `main` in `find.c` is written in such a way that `find` returns `0` if the needle is found, else it returns `1`. You can check the so-called "exit code" with which `main` returns by executing

```
echo $?
```

after running some other command. For instance, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2008
echo $?
```

you should see `0`, since `2008` is, again, among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search` (written by you) should return `true`, in which case `main` (written by us) should return (i.e., exit with) `0`. By contrast, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2013
echo $?
```

you should see `1`, since `2013` is, again, not among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search` (written by you) should return `false`, in which case `main` (written by us) should return (i.e., exit with) `1`. Make sense?

When ready to check the correctness of your program officially with `check50`, you may execute the below. Be sure to run the command inside of `~/Dropbox/hacker3/find`.

```
check50 2014/x/hacker3/find helpers.c
```

Incidentally, be sure not to get into the habit of testing your code with `check50` before testing it yourself. (And definitely don't get into an even worse habit of only testing your code with `check50`!) Suffice it to say `check50` doesn't exist in the real world, so running your code with your own sample inputs, comparing actual output against expected output, is the best habit to get into sooner rather than later. Truly, don't do yourself a long-term disservice!

Anyhow, if you'd like to play with the staff's own implementation of `find` in the appliance, you may execute the below.

```
~cs50/hacker3/find
```

Sorting

- Alright, linear search is pretty meh. Recall from Week 0 and Week 3 that we can do better, but first we'd best sort that hay.
- Notice that `find.c` calls `sort`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully too in `helpers.c`! Take a peek at `helpers.c` with `gedit`, and you'll see that `sort` returns immediately, even though `find`'s `main` function does pass it an actual array.

  Now, recall the syntax for declaring an array. Not only do you specify the array's type, you also specify its size between brackets, just as we do for `haystack` in `find.c`:

```
int haystack[MAX];
```

  But when passing an array, you only specify its name, just as we do when passing `haystack` to `sort` in `find.c`:

```
sort(haystack, size);
```

  (Why do we also pass in the size of that array separately?)

  When declaring a function that takes a one-dimensional array as an argument, though, you don't need to specify the array's size, just as we don't when declaring `sort` in `helpers.h` (and `helpers.c`):

```
void sort(int values[], int n);
```

  Go ahead and implement `sort` so that the function actually sorts, from smallest to largest, the array of numbers that it's passed, in such a way that its running time is in $O(n)$, where n is the array's size. Yes, this running time is possible because you may assume that each of the array's numbers wil be non-negative and less than `LIMIT`, a constant defined in `generate.c`. Leverage that assumption! However, realize that the array might contain duplicates.

  A previous version of this specification accidentally asked for a running time of $O(n^2)$ instead of $O(n)$, so we will accept either. But you are encouraged to (re-)try to achieve $O(n)$, since that's the intended challenge!

  Now, technically, because we've bounded with a constant the amount of hay that `find` will accept (and because the value of `sort`'s second parameter is bounded by an `int`'s finitely many bits), the running time of `sort`, however implemented, is arguably $O(1)$. Even so, for the sake of this asymptotic challenge, think of the size of `sort`'s input as n.

Anyhow, take care not to alter our declaration of `sort`. Its prototype must remain:

```
void sort(int values[], int n);
```

As this return type of `void` implies, this function must not return a sorted array; it must instead "destructively" sort the actual array that it's passed by moving around the values therein. As we'll discuss in Week 4, arrays are not passed "by value" but instead "by reference," which means that `sort` will not be passed a copy of an array but, rather, the original array itself.

Although you may not alter our declaration of `sort`, you're welcome to define your own function(s) in `helpers.c` that `sort` itself may then call.

We leave it to you to determine how best to test your implementation of `sort`. But don't forget that `printf` and, per Week 3's first lecture, `gdb` are your friends. And don't forget that you can generate the same sequence of pseudorandom numbers again and again by explicitly specifying `generate`'s seed. Before you ultimately submit, though, be sure to remove any such calls to `printf`, as we like our programs' outputs just they way they are!

Incidentally, check out **Resources** on the course's website for a quick-reference guide for `gdb`. If you'd like to play with the staff's own implementation of `find` in the appliance, you may execute the below.

```
~cs50/hacker3/find
```

No `check50` for this one!

- Now that `sort` (presumably) works, it's time to improve upon `search`, the other function that lives in `helpers.c`. Recall that your first version implemented linear search. Rip out the lines that you wrote earlier (sniff) and re-implement `search` as Binary Search, that divide-and-conquer strategy that we employed in Week 0 and again in Week 3. You are welcome to take an iterative or, per Week 4, a recursive approach. If you pursue the latter, though, know that you may not change our declaration of `search`, but you may write a new, recursive function (that perhaps takes different parameters) that `search` itself calls. When it comes time to submit this problem set, it suffices to submit this new-and-improved version of `search`; you needn't submit your original version that used linear search.

The Game Begins

- And now it's time to play. The Game of Fifteen is a puzzle played on a square, two-dimensional board with numbered tiles that slide. The goal of this puzzle is to arrange the board's tiles from smallest to largest, left to right, top to bottom, with an empty space in board's bottom-right corner, as in the below.

Sliding any tile that borders the board's empty space in that space constitutes a "move." Although the configuration above depicts a game already won, notice how the tile numbered 12 or the tile numbered 15 could be slid into the empty space. Tiles may not be moved diagonally, though, or forcibly removed from the board.

Although other configurations are possible, we shall assume that this game begins with the board's tiles in reverse order, from largest to smallest, left to right, top to bottom, with an empty space in the board's bottom-right corner. If, however, and only if the board contains an odd number of tiles (i.e., the height and width of the board are even), the positions of tiles numbered 1 and 2 must be swapped, as in the below. The puzzle is solvable from this configuration.

- Navigate your way to `~/Dropbox/hacker3/fifteen/`, and take a look at `fifteen.c` with `gedit`. (Remember how?) Within this file is an entire framework for the Game of Fifteen. The challenge up next is to complete this game's implementation.

  Implement God Mode for this game.

  First implement `init` in such a way that the board is initialized to a pseudorandom but solvable configuration. **To be clear, whereas the standard edition of this problem set requires that the board be initialized to a specific configuration, this Hacker Edition requires that it be initialized to a pseudorandom but still solvable configuration.** Then complete the implementation of `draw`, `move`, and `won` so that a human can actually play the game. But embed in the game a cheat, whereby, rather than typing an integer betwen 1 and d^2 - 1, where d is the board's dimension, the human can also type

```
GOD
```

  to compel "the computer" to take control of the game and solve it (using any strategy, optimal or non-optimal), making, say, only four moves per second so that the human can actually watch. Presumably, you'll need to swap out `GetInt` for something more versatile. It's fine if your implementation of God Mode only works (bearably fast) for $d \leq 4$; you need not worry about testing God Mode for $d > 4$. Oh and you can't implement God Mode by remembering how `init` initialized the board (as by remembering the sequence of moves that got your program to some pseudorandom but solvable state). That'd be, um, cheating. At cheating.

  To test your implementation, you can certainly try playing it yourself, with or without God Mode enabled. (Know that you can quit your program by hitting ctrl-c.) Be sure that you (and we) cannot crash your program, as by providing bogus tile numbers. And know that, much like you automated input into `find`, so can you automate execution of this game via input redirection if you store in some file a winning sequence of moves for some configuration.

Any design decisions not explicitly prescribed herein (e.g., how much space you should leave between numbers when printing the board) are intentionally left to you. Presumably the board, when printed, should look something like the below (albeit pseudorandom), but we leave it to you to implement your own vision.

```
15 14 13 12

11 10  9  8

 7  6  5  4

 3  1  2  _
```

Incidentally, recall that the positions of tiles numbered `1` and `2` should only be swapped (as they are in the 4 × 4 example above) if the board has an odd number of tiles (as does the 4 × 4 example above). If the board has an even number of tiles, those positions should not be swapped. Consider, for instance, the 3 × 3 example below:

```
8  7  6

5  4  3

2  1  _
```

Feel free to tweak the appropriate argument to usleep to speed up animation. In fact, you're welcome to alter the aesthetics of the game. For (optional) fun with "ANSI escape sequences," including color, take a look at our implementation of `clear` and check outhttp://www.isthe.com/chongo/tech/comp/ansi_escapes.html for more tricks.

You're welcome to write your own functions and even change the prototypes of functions we wrote. But we ask that you not alter the flow of logic in `main` so that we can automate some tests of your program. In particular, `main` must only return `0` if and when the user has actually won the game; non-zero values should be returned in any cases of error, as implied by our distribution code. And be sure not to alter the staff's implementation `save` or `main`'s usage thereof. If in doubt as to whether some design decision of yours might run counter to the staff's wishes, simply contact your teaching fellow.

If you'd like to play with the staff's own implementation of `fifteen` in the appliance, including God Mode, you may execute the below.

```
~cs50/hacker3/fifteen
```

Speaking of God Mode, where to begin? Well, first read up on this Game of Fifteen. Wikipedia is probably a good starting point:

http://en.wikipedia.org/wiki/N-puzzle

Then dive a bit deeper, perhaps reading up on an algorithm called A*.

http://en.wikipedia.org/wiki/A*_search_algorithm

Consider using "Manhattan distance" (aka "city-block distance") as your implementation's heuristic. If you find that A* takes up too much memory (particularly for $d \geq 4$), though, you might want to take a look at iterative deepening A* (IDA*) instead:

http://webdocs.cs.ualberta.ca/~tony/RecentPapers/pami94.pdf

The staff's own implementation, meanwhile, utilizes an algorithm like that in this paper:

http://larc.unt.edu/ian/pubs/saml.pdf

You're welcome to expand your search for ideas beyond those in these papers, but take care that your research does not lead you to actual code. Curling up with others' pseudocode is fine, but do click away if you stumble upon actual implementations (whether in C or other languages).

Alright, get to it, implement this game!

Problem Set 3: Game of Fifteen

This is the Hacker Edition of Problem Set 3. It cannot be submitted for credit.

Objectives

- Introduce you to larger programs and programs with multiple source files.
- Accustom you to reading someone else's code.
- Empower you with Makefiles.

- Introduce you to literature in computer science.
- Implement a party favor.

Recommended Reading

- Page 17 of http://www.howstuffworks.com/c.htm.
- Chapters 13, 15, and 18 of *Programming in C*.

diff pset3 hacker3

- Hacker Edition dares you to implement sort in $O(n)$ instead of $O(n^2)$.
- Hacker Edition asks you to play God.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.

- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 3's shorts and watch the shorts on bubble sort, insertion sort, and selection sort. Then head to Week 4's shorts and watch the short on `gdb`. (Phew, so many shorts! And so many sorts! Ha.)
  - `gdb` lets you "debug" program, but, more specifically, what does it let you do?
  - Why does binary search require that an array be sorted?
  - Why is bubble sort in $O(n^2)$?
  - Why is insertion sort in $\Omega(n)$?
  - What's the worst-case running time of merge sort?
  - In no more than 3 sentences, how does selection sort work?

Getting Started

- Recall that, for Problem Sets 1 and 2, you started writing programs from scratch, creating your own `pset1` and `pset2` directories with `mkdir`. For Problem Set 3, you'll instead download "distribution code" (otherwise known as a "distro"), written by us, and add your own lines of code to it. You'll first need to read and understand our code, though, so this problem set is as much about learning to read someone else's code as it is about writing your own!

  Let's get you started. Go ahead and open a terminal window if not open already (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

```
update50
```

  to make sure your appliance is up-to-date. Then execute

```
cd ~/Dropbox
```

  followed by

```
wget http://cdn.cs50.net/2013/fall/psets/3/hacker3/hacker3.zip
```

  to download a ZIP of this problem set's distro into your appliance (with a command-line program called `wget`). You should see a bunch of output followed by:

```
'hacker3.zip' saved
```

  If you instead see

```
unable to resolve host address
```

your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

Ultimately, confirm that you've indeed downloaded `hacker3.zip` by executing:

```
ls
```

Then, run

```
unzip hacker3.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `hacker3` as well. Proceed to execute

```
cd hacker3
```

followed by

```
ls
```

and you should see that the directory contains two "subdirectories":

```
fifteen  find
```

Fun times ahead!

Searching

- Okay, let's dive into the first of those subdirectories. Execute the command below in a terminal window in your appliance.

```
cd ~/hacker3/find/
```

If you list the contents of this directory, you should see the below.

```
helpers.c  helpers.h  Makefile  find.c  generate.c
```

Wow, that's a lot of files, eh? Not to worry, we'll walk you through them.

- Implemented in `generate.c` is a program that uses a "pseudorandom-number generator" (via a function called `rand`) to generate a whole bunch of random (well, pseudorandom, since computers can't actually generate truly random) numbers, one per line. (Cf.https://www.cs50.net/resources/cppreference.com/stdother/rand.html.) Go ahead and compile this program by executing the command below.

```
make generate
```

Now run the program you just compiled by executing the command below.

```
./generate
```

You should be informed of the program's proper usage, per the below.

```
Usage: generate n [s]
```

As this output suggests, this program expects one or two command-line arguments. The first, `n`, is required; it indicates how many pseudorandom numbers you'd like to generate. The second, `s`, is optional, as the brackets are meant to imply; if supplied, it represents the value that the pseudorandom-number generator should use as its "seed." A seed is simply an input to a pseudorandom-number generator that influences its outputs. For instance, if you seed `rand` by first calling `srand` (another function whose purpose is to "seed"`rand`) with an argument of, say, `1`, and then call `rand` itself three times, `rand` might return `17767`, then `9158`, then `39017`. (Cf.https://www.cs50.net/resources/cppreference.com/stdother/srand.html.) But if you instead seed `rand` by first calling `srand` with an argument of, say, `2`, and then call `rand` itself three times, `rand` might instead return `38906`, then `31103`, then `52464`. But if you re-seed`rand` by calling `srand` again with an argument of `1`, the next three times you call `rand`, you'll again get `17767`, then `9158`, then `39017`! See, not so random.

Go ahead and run this program again, this time with a value of, say, `10` for `n`, as in the below; you should see a list of 10 pseudorandom numbers.

```
./generate 10
```

Run the program a third time using that same value for n; you should see a different list of 10 numbers. Now try running the program with a value for s too (e.g., 0), as in the below.

```
./generate 10 0
```

Now run that same command again:

```
./generate 10 0
```

Bet you saw the same "random" sequence of ten numbers again? Yup, that's what happens if you don't vary a pseudorandom number generator's initial seed.

- Now take a look at `generate.c` itself with `gedit`. (Remember how?) Comments atop that file explain the program's overall functionality. But it looks like we forgot to comment the code itself. Read over the code carefully until you understand each line and then comment our code for us, replacing each `TODO` with a phrase that describes the purpose or functionality of the corresponding line(s) of code. (Know that an `unsigned int` is just an `int` that cannot be negative.) And for more details on `rand` and `srand`, recall that you can execute:

- `man rand`

```
man srand
```

Once done commenting `generate.c`, re-compile the program to be sure you didn't break anything by re-executing the command below.

```
make generate
```

If `generate` no longer compiles properly, take a moment to fix what you broke!

Now, recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. Notice, in fact, how `make` just executed a pretty long command for you, per the tool's output. However, as your programs grow in size, `make` won't be able to infer from context anymore how to compile your code; you'll need to start telling `make` how to compile your program, particularly when they involve multiple source (i.e., `.c`) files. And so we'll start relying on "Makefiles," configuration files that tell `make` exactly what to do.

How did `make` know how to compile generate in this case? It actually used a configuration file that we wrote. Using `gedit`, go ahead and look at the file called `Makefile` that's in the same directory as `generate.c`. This `Makefile` is essentially a list of rules that we wrote for you that tells make how to build generate from `generate.c` for you. The relevant lines appear below.

```
generate: generate.c
    `clang` -ggdb -std=c99 -Wall -Werror -o generate generate.c
```

The first line tells `make` that the "target" called `generate` should be built by invoking the second line's command. Moreover, that first line tells `make` that `generate` is dependent on `generate.c`, the implication of which is that `make` will only re-build generate on subsequent runs if that file was modified since `make` last built `generate`. Neat time-saving trick, eh? In fact, go ahead and execute the command below again, assuming you haven't modified `generate.c`.

```
make generate
```

You should be informed that generate is already up to date. Incidentally, know that the leading whitespace on that second line is not a sequence of spaces but, rather, a tab. Unfortunately, `make` requires that commands be preceded by tabs, so be careful not to change them to spaces with `gedit` (which automatically converts tabs to four spaces), else you may encounter strange errors! The `-Werror` flag, recall, tells `clang` to treat warnings (bad) as though they're errors (worse) so that you're forced (in a good, instructive way!) to fix them.

- Now take a look at `find.c` with `gedit`. Notice that this program expects a single command-line argument: a "needle" to search for in a "haystack" of values. Once done looking over the code, go ahead and compile the program by executing the command below.

```
make find
```

Notice, per that command's output, that `make` actually executed the below for you.

```
clang -ggdb -std=c99 -Wall -Werror -o find find.c helpers.c -lcs50 -
lm
```

Notice further that you just compiled a program comprising not one but two `.c` files: `helpers.c` and `find.c`. How did `make` know what to do? Well, again, open up `Makefile` to see the man behind the curtain. The relevant lines appear below.

```
find: find.c helpers.c helpers.h
    clang -ggdb -std=c99 -Wall -Werror -o find find.c helpers.c -
lcs50 -lm
```

Per the dependencies implied above (after the colon), any changes to `find.c`, `helpers.c`, or `helpers.h` will compel `make` to rebuild find the next time it's invoked for this target.

Go ahead and run this program by executing, say, the below.

```
./find 13
```

You'll be prompted to provide some hay (i.e., some integers), one "straw" at a time. As soon as you tire of providing integers, hit ctrl-d to send the program an `EOF` (end-of-file) character. That character will compel `GetInt` from the CS50 Library to return `INT_MAX`, a constant that, per `find.c`, will compel `find` to stop prompting for hay. The program will then look for that needle in the hay you provided, ultimately reporting whether the former was found in the latter. In short, this program searches an array for some value. At least, it should, but it won't find anything yet! That's where you come in. More on your role in a bit.

In turns out you can automate this process of providing hay, though, by "piping" the output of `generate` into `find` as input. For instance, the command below passes 1,000 pseudorandom numbers to `find`, which then searches those values for `42`.

```
./generate 1000 | ./find 42
```

Note that, when piping output from `generate` into `find` in this manner, you won't actually see `generate`'s numbers, but you will see `find`'s prompts.

Alternatively, you can "redirect" `generate`'s output to a file with a command like the below.

```
./generate 1000 > numbers.txt
```

You can then redirect that file's contents as input to `find` with the command below.

```
./find 42 < numbers.txt
```

Let's finish looking at that `Makefile`. Notice the line below.

```
all: find generate
```

This target implies that you can build both `generate` and `find` simply by executing the below.

```
make all
```

Even better, the below is equivalent (because `make` builds a `Makefile`'s first target by default).

```
make
```

If only you could whittle this whole problem set down to a single command! Finally, notice these last lines in `Makefile`:

```
clean:
    rm -f *.o a.out core find generate
```

This target allows you to delete all files ending in `.o` or called `core` (more on that soon!), `find`, or `generate` simply by executing the command below.

```
make clean
```

Be careful not to add, say, `*.c` to that last line in `Makefile`! (Why?) Any line, incidentally, that begins with `#` is just a comment.

- And now the fun begins! Notice that `find.c` calls `search`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully in `helpers.c`! (To be sure, we could have put the contents of `helpers.h` and `helpers.c` in `find.c` itself. But it's sometimes better to organize programs into multiple files, especially when some functions are essentially utility functions that might later prove useful to other programs as well, much like those in the CS50 Library.) Take a peek at `helpers.c` with `gedit`, and you'll see that `search` always returns `false`, whether or not `value` is in `values`. Re-write `search` in such a way that it uses linear search, returning `true` if `value` is in `values` and `false` if `value` is not in `values`. Take care to return `false` right away if `n` isn't even positive.

  When ready to check the correctness of your program, try running the command below.

```
./generate 1000 50 | ./find 2008
```

Because one of the numbers outputted by `generate`, when seeded with `50`, is `2008`, your code should find that "needle"! By contrast, try running the command below as well.

```
./generate 1000 50 | ./find 2013
```

Because `2013` is not among the numbers outputted by `generate`, when seeded with `50`, your code shouldn't find that needle. Best to try some other tests as well, as by running `generate` with some seed, taking a look at its output, then piping that same output to `find`, looking for a "needle" you know to be among the "hay".

Incidentally, note that `main` in `find.c` is written in such a way that `find` returns `0` if the needle is found, else it returns `1`. You can check the so-called "exit code" with which `main` returns by executing

```
echo $?
```

after running some other command. For instance, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2008
echo $?
```

you should see `0`, since `2008` is, again, among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search` (written by you) should return `true`, in which case `main` (written by us) should return (i.e., exit with) `0`. By contrast, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2013
echo $?
```

you should see `1`, since `2013` is, again, not among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search` (written by you) should return `false`, in which case `main` (written by us) should return (i.e., exit with) `1`. Make sense?

When ready to check the correctness of your program officially with `check50`, you may execute the below. Be sure to run the command inside of `~/Dropbox/hacker3/find`.

```
check50 2014/x/hacker3/find helpers.c
```

Incidentally, be sure not to get into the habit of testing your code with `check50` before testing it yourself. (And definitely don't get into an even worse habit of only testing your code with `check50`!) Suffice it to say `check50` doesn't exist in the real world, so running your code with your own sample inputs, comparing actual output against expected output, is the best habit to get into sooner rather than later. Truly, don't do yourself a long-term disservice!

Anyhow, if you'd like to play with the staff's own implementation of `find` in the appliance, you may execute the below.

```
~cs50/hacker3/find
```

Sorting

- Alright, linear search is pretty meh. Recall from Week 0 and Week 3 that we can do better, but first we'd best sort that hay.

- Notice that `find.c` calls `sort`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully too in `helpers.c`! Take a peek at `helpers.c` with `gedit`, and you'll see that `sort` returns immediately, even though `find`'s `main` function does pass it an actual array.

  Now, recall the syntax for declaring an array. Not only do you specify the array's type, you also specify its size between brackets, just as we do for `haystack` in `find.c`:

```
int haystack[MAX];
```

  But when passing an array, you only specify its name, just as we do when passing `haystack` to `sort` in `find.c`:

```
sort(haystack, size);
```

  (Why do we also pass in the size of that array separately?)

  When declaring a function that takes a one-dimensional array as an argument, though, you don't need to specify the array's size, just as we don't when declaring `sort` in `helpers.h` (and `helpers.c`):

```
void sort(int values[], int n);
```

  Go ahead and implement `sort` so that the function actually sorts, from smallest to largest, the array of numbers that it's passed, in such a way that its running time is in $O(n)$, where n is the array's size. Yes, this running time is possible because you may assume that each of the array's numbers wil be non-negative and less than `LIMIT`, a constant defined in `generate.c`. Leverage that assumption! However, realize that the array might contain duplicates.

  A previous version of this specification accidentally asked for a running time of $O(n^2)$ instead of $O(n)$, so we will accept either. But you are encouraged to (re-)try to achieve $O(n)$, since that's the intended challenge!

  Now, technically, because we've bounded with a constant the amount of hay that `find` will accept (and because the value of `sort`'s second parameter is bounded by an `int`'s finitely many bits), the running time of `sort`, however implemented, is arguably $O(1)$. Even so, for the sake of this asymptotic challenge, think of the size of `sort`'s input as n.

  Anyhow, take care not to alter our declaration of `sort`. Its prototype must remain:

```
void sort(int values[], int n);
```

As this return type of `void` implies, this function must not return a sorted array; it must instead "destructively" sort the actual array that it's passed by moving around the values therein. As we'll discuss in Week 4, arrays are not passed "by value" but instead "by reference," which means that `sort` will not be passed a copy of an array but, rather, the original array itself.

Although you may not alter our declaration of `sort`, you're welcome to define your own function(s) in `helpers.c` that `sort` itself may then call.

We leave it to you to determine how best to test your implementation of `sort`. But don't forget that `printf` and, per Week 3's first lecture, `gdb` are your friends. And don't forget that you can generate the same sequence of pseudorandom numbers again and again by explicitly specifying `generate`'s seed. Before you ultimately submit, though, be sure to remove any such calls to `printf`, as we like our programs' outputs just they way they are!

Incidentally, check out **Resources** on the course's website for a quick-reference guide for `gdb`. If you'd like to play with the staff's own implementation of `find` in the appliance, you may execute the below.

```
~cs50/hacker3/find
```

No `check50` for this one!

- Now that `sort` (presumably) works, it's time to improve upon `search`, the other function that lives in `helpers.c`. Recall that your first version implemented linear search. Rip out the lines that you wrote earlier (sniff) and re-implement `search` as Binary Search, that divide-and-conquer strategy that we employed in Week 0 and again in Week 3. You are welcome to take an iterative or, per Week 4, a recursive approach. If you pursue the latter, though, know that you may not change our declaration of `search`, but you may write a new, recursive function (that perhaps takes different parameters) that `search` itself calls. When it comes time to submit this problem set, it suffices to submit this new-and-improved version of `search`; you needn't submit your original version that used linear search.

The Game Begins

- And now it's time to play. The Game of Fifteen is a puzzle played on a square, two-dimensional board with numbered tiles that slide. The goal of this puzzle is to arrange the board's tiles from smallest to largest, left to right, top to bottom, with an empty space in board's bottom-right corner, as in the below.

Sliding any tile that borders the board's empty space in that space constitutes a "move." Although the configuration above depicts a game already won, notice how the tile numbered

12 or the tile numbered 15 could be slid into the empty space. Tiles may not be moved diagonally, though, or forcibly removed from the board.

Although other configurations are possible, we shall assume that this game begins with the board's tiles in reverse order, from largest to smallest, left to right, top to bottom, with an empty space in the board's bottom-right corner. If, however, and only if the board contains an odd number of tiles (i.e., the height and width of the board are even), the positions of tiles numbered 1 and 2 must be swapped, as in the below. The puzzle is solvable from this configuration.

- Navigate your way to `~/Dropbox/hacker3/fifteen/`, and take a look at `fifteen.c` with `gedit`. (Remember how?) Within this file is an entire framework for the Game of Fifteen. The challenge up next is to complete this game's implementation.

  Implement God Mode for this game.

  First implement `init` in such a way that the board is initialized to a pseudorandom but solvable configuration. **To be clear, whereas the standard edition of this problem set requires that the board be initialized to a specific configuration, this Hacker Edition requires that it be initialized to a pseudorandom but still solvable configuration.** Then complete the implementation of `draw`, `move`, and `won` so that a human can actually play the game. But embed in the game a cheat, whereby, rather than typing an integer betwen 1 and d^2 - 1, where d is the board's dimension, the human can also type

```
GOD
```

  to compel "the computer" to take control of the game and solve it (using any strategy, optimal or non-optimal), making, say, only four moves per second so that the human can actually watch. Presumably, you'll need to swap out `GetInt` for something more versatile. It's fine if your implementation of God Mode only works (bearably fast) for $d \leq 4$; you need not worry about testing God Mode for $d > 4$. Oh and you can't implement God Mode by remembering how `init` initialized the board (as by remembering the sequence of moves that got your program to some pseudorandom but solvable state). That'd be, um, cheating. At cheating.

  To test your implementation, you can certainly try playing it yourself, with or without God Mode enabled. (Know that you can quit your program by hitting ctrl-c.) Be sure that you (and we) cannot crash your program, as by providing bogus tile numbers. And know that, much like you automated input into `find`, so can you automate execution of this game via input redirection if you store in some file a winning sequence of moves for some configuration.

  Any design decisions not explicitly prescribed herein (e.g., how much space you should leave between numbers when printing the board) are intentionally left to you. Presumably the board, when printed, should look something like the below (albeit pseudorandom), but we leave it to you to implement your own vision.

```
15 14 13 12

11 10  9  8

 7  6  5  4

 3  1  2  _
```

Incidentally, recall that the positions of tiles numbered `1` and `2` should only be swapped (as they are in the 4 × 4 example above) if the board has an odd number of tiles (as does the 4 × 4 example above). If the board has an even number of tiles, those positions should not be swapped. Consider, for instance, the 3 × 3 example below:

```
8  7  6

5  4  3

2  1  _
```

Feel free to tweak the appropriate argument to usleep to speed up animation. In fact, you're welcome to alter the aesthetics of the game. For (optional) fun with "ANSI escape sequences," including color, take a look at our implementation of `clear` and check outhttp://www.isthe.com/chongo/tech/comp/ansi_escapes.html for more tricks.

You're welcome to write your own functions and even change the prototypes of functions we wrote. But we ask that you not alter the flow of logic in `main` so that we can automate some tests of your program. In particular, `main` must only return `0` if and when the user has actually won the game; non-zero values should be returned in any cases of error, as implied by our distribution code. And be sure not to alter the staff's implementation `save` or `main`'s usage thereof. If in doubt as to whether some design decision of yours might run counter to the staff's wishes, simply contact your teaching fellow.

If you'd like to play with the staff's own implementation of `fifteen` in the appliance, including God Mode, you may execute the below.

```
~cs50/hacker3/fifteen
```

Speaking of God Mode, where to begin? Well, first read up on this Game of Fifteen. Wikipedia is probably a good starting point:

http://en.wikipedia.org/wiki/N-puzzle

Then dive a bit deeper, perhaps reading up on an algorithm called A*.

http://en.wikipedia.org/wiki/A*_search_algorithm

Consider using "Manhattan distance" (aka "city-block distance") as your implementation's heuristic. If you find that A* takes up too much memory (particularly for $d \geq 4$), though, you might want to take a look at iterative deepening A* (IDA*) instead:

http://webdocs.cs.ualberta.ca/~tony/RecentPapers/pami94.pdf

The staff's own implementation, meanwhile, utilizes an algorithm like that in this paper:

http://larc.unt.edu/ian/pubs/saml.pdf

You're welcome to expand your search for ideas beyond those in these papers, but take care that your research does not lead you to actual code. Curling up with others' pseudocode is fine, but do click away if you stumble upon actual implementations (whether in C or other languages).

Alright, get to it, implement this game!

Problem Set 4: Breakout

This is the Hacker Edition of Problem Set 4. It cannot be submitted for credit.

with thanks to Eric Roberts of Stanford

Objectives

- Learn an API.
- Build a game with a real GUI.
- Acquaint you with event handling.
- Impress your friends.

diff pset4 hacker4

- Hacker Edition requires that ball bounce off paddle at varying angles.
- Hacker Edition offers bonus features, including:
  - God Mode;

- variable scoring;
- shrinking paddles;
- accelerating velocity; and/or
- lasers.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.

- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 4's shorts and watch the shorts on pointers and strings. Then head to Week 5's shorts and watch the short on the CS50 Library.

Backstory

One day in the late summer of 1975, Nolan Bushnell [founder of Atari and, um, Chuck E. Cheese's], defying the prevailing wisdom that paddle games were over, decided to develop a single-player version of Pong; instead of competing against an opponent, the player would volley the ball into a wall that lost a brick whenever it was hit. He called [Steve] Jobs into his office, sketched it out on his little blackboard, and asked him to design it. There would be a bonus,

Bushnell told him, for every chip fewer than fifty that he used. Bushnell knew that Jobs was not a great engineer, but he assumed, correctly, that he would recruit [Steve] Wozniak, who was always hanging around. "I looked at it as a two-for-one thing," Bushnell recalled. "Woz was a better engineer."

Wozniak was thrilled when Jobs asked him to help and proposed splitting the fee. "This was the most wonderful offer in my life, to actually design a game that people would use," he recalled. Jobs said it had to be done in four days and with the fewest chips possible. What he hid from Wozniak was that the deadline was one that Jobs had imposed, because he needed to get to the All One Farm to help prepare for the apple harvest. He also didn't mention that there was a bonus tied to keeping down the number of chips.

"A game like this might take most engineers a few months," Wozniak recalled. "I thought that there was no way I could do it, but Steve made me sure that I could." So he stayed up four nights in a row and did it. During the day at HP, Wozniak would sketch out his design on paper. Then, after a fast-food meal, he would go right to Atari and stay all night. As Wozniak churned out the design, Jobs sat on a bench to his left implementing it by wire-wrapping the chips onto a breadboard. "While Steve was breadboarding, I spent time playing my favorite game ever, which was the auto racing game Gran Trak 10," Wozniak said.

Astonishingly, they were able to get the job done in four days, and Wozniak used only forty-five chips. Recollections differ, but by most accounts Jobs simply gave Wozniak half of the base fee and not the bonus Bushnell paid for saving five chips. It would be another ten years before Wozniak discovered (by being shown the tale in a book on the history of Atari titled *Zap*) that Jobs had been paid this bonus….
Steve Jobs— Walter Isaacson

Getting Started

Your challenge for this problem set is to implement the same game that Steve and Steve did, albeit in software rather than hardware. That game is Breakout.

Now, Problem Set 3 was also a game, but its graphical user interface (GUI) wasn't exactly a GUI; it was more of a textual user interface, since we essentially simulated graphics with `printf`. Let's give Breakout an actual GUI by building atop the Stanford Portable Library (SPL), which is similar in spirit to the CS50 Library but includes an API (application programming interface) for GUI programming and more.

Let's get you started.

- As always, first open a terminal window and execute

  ```
  update50
  ```

  to make sure your appliance is up-to-date.

- Next execute

```
cd ~/Dropbox
```

  followed by

```
wget http://cdn.cs50.net/2013/fall/lectures/5/m/src5m.zip
```

  to download some source code from Week 5. If you instead see

```
unable to resolve host address
```

  your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

  When ready, unzip the file with

```
unzip src5m.zip
```

  at which point you should find yourself with a directory called `src5m` in `~/Dropbox`. Navigate your way into it with

```
cd src5m
```

  and then execute

```
ls
```

  and you should see the below.

```
bounce.c  checkbox.c  cursor.c  Makefile  spl      text.c
button.c  click.c     label.c   slider.c  spl.jar  window.c
```

  Go ahead and compile all of these programs at once (thanks to the `Makefile` in there) by executing the below.

```
make
```

  Then execute the simplest of those programs as follows.

```
./window
```

A window quite like the below should appear and then disappear after 5 seconds.

Neat, eh? Open up `window.c` with `gedit`. Let's now take a tour.

How did we know how to call `newGWindow` like that? Well, there aren't `man` pages for SPL, but you can peruse the relevant header file (`gwindow.h`). In fact, notice that inside of `src5m` is a subdirectory called `spl`. Inside of that is another subdirectory called `include`. If you take a look there, you'll find `gwindow.h`. Open it up with `gedit` and look around. (Alternatively, you can see it athttp://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/include/gwindow.h.) Hm, a bit overwhelming. But because SPL's author has commented the code in a standard way, it turns out that you can generate more user-friendly, web-based documentation as a result! Indeed, take a look now at http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gwindow.html, and you'll see a much friendlier format. Click `newGWindow` under **Functions**, and you'll see its prototype:

```
GWindow newGWindow(double width, double height);
```

That's how we knew! See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/index.html for an index into SPL's documentation, though we'll point out more specific places to look.

Now, in the interests of full disclosure, we should mention that SPL is still in beta, so there may be some bugs in its documentation. When in doubt, best to consult those raw header files instead!

- Next open up `click.c` with `gedit`. This one's a bit more involved but it's representative of how to "listen" for "events", quite like those you could "broadcast" in Scratch. Let's take a look.

  See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gevents.html for SPL's documentation of `GEvent`.

- Now open up `cursor.c` with `gedit`. This program, too, handles events, but it also responds to those events by moving a circle (well, a `GOval`) in lockstep. Let's take a look.

  See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gobjects.html for SPL's documentation of `GOval` and other types of objects.

- Next open `bounce.c` with `gedit`. This one uses a bit of arithmetic to bounce a circle back and forth between a window's edges. Let's take a look.

- Now take a look at `button.c`, `checkbox.c`, `label.c`, `slider.c`, and `text.c` in any order with `gedit`. No videos for those, but do try running them each to see what more you can do with SPL.

- Okay, now that you've been exposed to a bit of GUI programming, let's turn our attention to this problem set's own distribution code. In a terminal window, execute

```
cd ~/Dropbox
```

  if not already there, and then execute

```
wget http://cdn.cs50.net/2013/fall/psets/4/hacker4/hacker4.zip
```

  to download a ZIP of this problem set's distro into your appliance. You should see a bunch of output followed by:

```
'hacker4.zip' saved
```

  As before, if you instead see

```
unable to resolve host address
```

  your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

  Ultimately, confirm that you've indeed downloaded `hacker4.zip` by executing:

```
ls
```

  Then, run

```
unzip hacker4.zip
```

  to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `hacker4` as well. Proceed to execute

```
cd hacker4
```

  followed by

```
ls
```

  and you should see that the directory contains three files and one subdirectory:

  - `Makefile`, which will let `make` know how to compile your program;
  - `breakout.c`, which contains a skeleton for your program;

    - `spl`, a subdirectory containing SPL; and
    - `spl.jar`, a "Java archive" that contains code (written in a language called Java) atop which SPL's C library is written.
- Let's see what the distribution code does. Go ahead and execute

```
make breakout
```

  or, more simply,

```
make
```

  to compile the distro. Then execute

```
./breakout
```

  to run the program as is. A window like the below should appear.

  Hm, not much of a game. Yet!
- Next try out the staff's solution (to the standard edition) by executing the below from within your own `~/Dropbox/hacker4` directory..

```
~cs50/pset4/breakout
```

  A window like the below should appear.

  How fun! Go ahead and click somewhere inside that window in order to play. The goal, quite simply, is to bounce the ball off of the paddle so as to hit bricks with it. If you break all the bricks, you win! But if you miss the ball three times, you lose! To quit the game, hit control-c back in the terminal window.

  Nice. Let's make your implementation look more like that one. But, first, a tour!
- Open up `breakout.c` with `gedit` and take a moment to scroll through it to get a sense of what lies ahead. It's a bit reminiscent of the skeleton for Game of Fifteen, no? But definitely some new functions in there, most from SPL. Let's walk through it from top to bottom.
    - Atop the file you'll see some familiar header files. We've also included `time.h` so that you have access to a "pseudorandom number generator" (PRNG), a function that can generate random (well, technically not-quite-random) numbers. We've also included some header files from SPL. Because those files are included in this problem set's distribution code (in `hacker4/spl/include`), we've used, as is required by C, double quotes (`"`)

around their filenames instead of the usual angled brackets (`<` and `>`) because they're not installed deep in the appliance itself.

- Next up are some constants, values that you don't need to change, but because the code we've written (and that you'll write) needs to know these values in a few places, we've factored them out as constants so that we or you could, theoretically, change them in one convenient location. By contrast, hard-coding the same number (pejoratively known as a "magic number") into your code in multiple places is considered bad practice, since you'd have to remember to change it, potentially, in all of those places.
- Below those constants are a bunch of prototypes for functions that are defined below `main`. More on each of those soon.
- Next up is our old friend, `main`. It looks like the first thing that `main` does is "seed" that so-called PRNG with the current time. (See `man srand48` and `man 2 time` if curious.) To seed a PRNG simply means to initialize it in such a way that the numbers it will eventually spit out will appear to be random. It's deliberate, then, that we're initializing the PRNG with the current time: time's always changing. Were we instead to initialize the PRNG with some hard-coded value, it'd always spit out the same sequence of "random" numbers.

  After that call to `srand48`, it looks like `main` calls `newGWindow`, passing in a desired `WIDTH` and `HEIGHT`. That function "instantiates" (i.e., creates) a new graphical window, returning some sort of reference thereto. (It's technically a pointer, but that detail, and the accompanying `*`, is, again, hidden from us by SPL.) That function's return value is apparently stored in a variable called `window` whose type is `GWindow`, which happens to be declared in a `gwindow.h` header file that you may have glimpsed earlier.

  Next, `main` calls `initBricks`, a function written partly by us (and, soon, mostly by you!) that instantiates a grid of bricks atop the game's window.

  Then `main` calls `initBall`, which instantiates the ball that will be used to play Breakout. Passed into that function is `window` so that the function knows where to "place" (i.e., draw) the ball. The function returns a `GOval` (graphical oval) whose width and height will simply be equal (ergo a circular ball).

  Called by `main` next is `initPaddle`, which instantiates the game's paddle; it returns a `GRect` (graphical rectangle).

  Then `main` calls `initScoreboard`, which instantiates the game's scoreboard, which is simply a `GLabel` (graphical label).

  Below all those function calls are a few definitions of variables, namely `bricks`, `lives`, and `points`. Below those is a loop, which is meant to iterate again and again so long as the user has lives left to live and bricks left to break. Of course, there's not much code in that loop now!

Below the loop is a call to `waitForClick`, a function that does exactly that so that the window doesn't close until the user intends.

Not too bad, right? Let's next take a closer look at those functions.

- In `initBricks`, you'll eventually write code that instantiates a grid of bricks in the window. Those constants we saw earlier, `ROWS` and `COLS`, represent that grid's dimensions. How to draw a grid of bricks on the screen? Well, odds are you'll want to employ a pair of `for` loops, one nested inside of the other. And within that innermost loop, you'll likely want to instantiate a `GRect` of some width and height (and color!) to represent a brick.
- In `initBall`, you'll eventually write code that instantiates a ball (that is, a circle, or really a `GOval`) and somehow center it in the window.
- In `initPaddle`, you'll eventually write code that instantiates a paddle (just a `GRect`) that's somehow centered in the bottom-middle of the game's window.
- Finally, in `initScoreboard`, you'll eventually write code that instantiates a scoreboard as, quite simply, a `GLabel` whose value is a number (well, technically, a `char*`, which we once knew as a `string`).
- Now, we've already implemented `updateScoreboard` for you. All that function does, given a `GWindow`, a `GLabel`, and an `int`, is convert the `int` to a `string` (okay, `char*`) using a function called `sprintf`, after which it sets the label to that value and then re-centers the label (in case the `int` has more digits than some previous `int`). Why did we allocate an array of size `12` for our representation of that `int` as a `string`? No worries if the reason's non-obvious, but give some though as to how wide the most positive (or most negative!) `int` might be. You're welcome to change this function, but you're not expected to.
- Last up is `detectCollision`, another function that we've written for you. (Phew!) This one's a bit more involved, so do spend some time reading through it. This function's purpose in life, given the ball as a `GOval`, is to determine whether that ball has collided with (i.e., is overlapping) some other object (well, `GObject`) in the game. (A `GRect`, `GOval`, or `GLabel` can also be thought of and treated as a `GObject`, per http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gobjects.html.) To do so, it cuts some corners (figuratively but also kind of literally) by checking whether any of the ball's "corners," as defined by the ball's "bounding box", per the below (wherein x and y represent coordinates, and r represents the ball's radius) are touching some other `GObject` (which might be a brick or a paddle or even something else).

Alright, ready to break out Breakout?

Breakout

Alright, if you're like me, odds are you'll find it easiest to implement Breakout via some baby steps, each of which will get you closer and closer to a great outcome. Rather than try to implement the whole game at once, allow me to suggest that you proceed as follows:

1. Try out the staff's solution again (via `~cs50/pset4/breakout` from within your own `~/Dropbox/pset4` directory) to remind yourself how our implementation behaves. Yours doesn't need to be identical. In fact, all the better if you personalize yours. But playing with our implementation should help guide you toward yours.
2. Implement `initPaddle`. Per the function's return value, your paddle should be implemented as a `GRect`. Odds are you'll first want to decide on a width and height for your paddle, perhaps declaring them both atop `breakout.c` with constants. Then calculate coordinates (*x* and *y*) for your paddle, keeping in mind that it should be initially aligned in the bottom-middle of your game's window. We leave it to you to decide exactly where. Odds are some arithmetic involving the window's width and height and the paddle's width and height will help you center it. Keep in mind that *x* and *y* refer to a `GRect`'s top-left corner, not its own middle. Your paddle's size and location doesn't need to match the staff's precisely, but it should be perfectly centered, near the window's bottom. You're welcome to choose a color for it too, for which `setColor` and `setFilled` might be of interest. Finally, instantiate your paddle with `newGRect`. (Take note of that function's prototype at http://cdn.cs50.net/2013/fall/psets/4/pset4/pset4/spl/doc/gobjects.html.) Then return the `GRect` returned by `newGRect` (rather than `NULL`, which the distribution code returns only so that the program will compile without `initPaddle` fully implemented).
3. Now, `initPaddle`'s purpose in life is only to instantiate and return a paddle (i.e., `GRect`). It shouldn't handle any of the paddle's movement. For that, turn your attention to the `TODO` up in `main`. Proceed to replace that `TODO` with some lines of code that respond to a user's mouse movements in such a way that the paddle follows the movements, but only along its (horizontal) x-axis. Look back at `cursor.c` for inspiration, but keep in mind that `cursor.c` allowed that circle to move along a (vertical) y-axis as well, which we don't want for Breakout, else the paddle could move anywhere (which might be cool but not exactly Breakout).
4. Now turn your attention to the `TODO` in `initBricks`. Implement that function in such a way that it instantiates a grid of bricks (with `ROWS` rows and `COLS` columns), with each such brick implemented as a `GRect`. Drawing a `GRect` (or even a bunch of them) isn't all that different from drawing a `GOval` (or circle). Odds are, though, you'll want to instantiate them within a `for` loop that's within a `for` loop. (Think back to `mario`, perhaps!) Be sure to leave a bit of a gap between adjacent bricks, just like we did; exactly how many pixels is up to you. And we leave it to you to select your bricks' colors.

5. Now implement `initBall`, whose purpose in life is to instantiate a ball in the window's center. (Another opportunity for a bit of arithmetic!) Per the function's prototype, be sure to return a `GOval`.

6. Then, back in `main`, where there used to be a `TODO`, proceed to write some additional code (within that same `while` loop) that compels that ball to move. Here, too, take baby steps. Look to `bounce.c` first for ideas on how to make the ball bounce back and forth between your window's edges. (Not the ultimate goal, but it's a step toward it!) Then figure out how to make the ball bounce up and down instead of left and right. (Closer!) Then figure out how to make the ball move at an angle. Then, utilize `drand48` to make the ball's initial velocity random, at least along its (horizontal) x-axis. Note that, per its `man` page, `drand48` returns "nonnegative double-precision floating-point values uniformly distributed between [0.0, 1.0)." In other words, it returns a `double` between 0.0 (inclusive) and 1.0 (exclusive). If you want your velocity to be faster than that, simply add some constant to it and/or multiply it by some constant!

   Ultimately, be sure that the ball still bounces off edges, including the window's bottom for now.

7. When ready, add some additional code to `main` (still somewhere inside of that `while` loop) that compels the ball to bounce off of the paddle if it collides with it on its way downward. **The angle at which the ball bounces off the paddle should depend on where the ball strikes the paddle; we leave it to you to decide on a formula.** Odds are you'll want to call that function we wrote, `detectCollision`, inside that loop in order to detect whether the ball's collided with something so that, if so, you can somehow handle such an event. Of course, the ball could collide with the paddle or with any one of those bricks. Keep in mind, then, that `detectCollision` could return any such `GObject`; it's left to you to determine what has been struck. Know, then, that if you store its return value, as with

```
GObject object = detectCollision(window, ball);
```

   you can determine whether that `object` is your game's paddle, as with the below.

```
if (object == paddle)
{
    // TODO
}
```

   More generally, you can determine if that `object` is a `GRect` with:

```
if (strcmp(getType(object), "GRect") == 0)
```

```
{
    // TODO
}
```

Once it comes time to add a `GLabel` to your game (for its scoreboard), you can similarly determine if that `object` is `GLabel`, in which case it might be a collision you want to ignore. (Unless you want your scoreboard to be something the ball can bounce off of. Ours isn't.)

```
if (strcmp(getType(object), "GLabel") == 0)
{
    // TODO
}
```

8. Once you have the ball bouncing off the paddle (and window's edges), focus your attention again on that `while` loop in `main` and figure out how to detect if the ball's hit a brick and how to remove that brick from the grid if so. Odds are you'll find `removeGWindow` of interest, per http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gwindow.html. **SPL's documentation incorrectly refers to that function as `remove`, but it's indeed `removeGWindow` you want, whose prototype, to be clear, is the below.**

```
void removeGWindow(GWindow gw, GObject gobj);
```

9. Now decide how to determine whether the ball has zoomed past the paddle and struck the window's bottom edge, in which case the user should lose a life and gameplay should probably pause until the user clicks the mouse button, as in the staff's implementation. Odds are detecting this situation isn't all that different from the code you already wrote for bouncing; you just don't want to bounce off that bottom edge anymore!

10. Next, implement `initScoreboard` in such a way that the function instantiates and positions a `GLabel` somewhere in your game's window. Then, enhance `main` in such a way that the text of that `GLabel` is updated with the user's score anytime the user breaks a brick. Indeed, be sure that your program keeps track of how many lives remain and how many bricks remain, the latter of which is inversely related to how many points you should give the user for each brick broken; our solution awards one point per brick, but you're welcome to offer different rewards. A user's game should end (i.e., the ball should stop moving) after a user runs out of lives or after all bricks are broken. We leave it to you to decide what to do in both cases, if anything more!

11. Lastly, amaze us by integrating at least two (2) of the following features into your game:

- Implement God Mode whereby, if the program is run with `./breakout GOD`, the game ignores the user's mouse movements and instead moves the paddle itself in perfect lockstep with the ball along its (horizontal) x-axis so that the ball never misses the paddle.
- Implement a shrinking-paddle mechanism whereby the paddle's width decreases as bricks are broken; we leave it to you to decide on a formula.
- Implement a variable-scoring mechanism whereby bricks higher in the game's grid are worth more points than are bricks lower in the game's grid; we leave it to you to decide on a formula.
- Implement a variable-velocity mechanism whereby the ball's velocity (along one or both axes) increases as bricks are broken; we leave it to you to decide on a formula.
- Implement lasers, whereby clicking the mouse button during gameplay results in the paddle shooting one or two laser beams upward toward bricks, whereby those beams can destroy them just like the ball can. However, if a beam strikes the ball itself, gameplay must end.

Because this game expects a human to play, no `check50` for this one! Best to invite some friends to find bugs!

Problem Set 5: Forensics

This is the Hacker Edition of Problem Set 5. It cannot be submitted for credit.

Objectives

- Acquaint you with file I/O.
- Get you more comfortable with data structures, hexadecimal, and pointers.
- Introduce you to computer scientists across campus.
- Help Mr. Boddy.

Recommended Reading*

- Chapters 9, 11, 14, and 16 of *Programming in C*
- http://www.cprogramming.com/tutorial/cfileio.html
- http://en.wikipedia.org/wiki/BMP_file_format
- http://en.wikipedia.org/wiki/Hexadecimal
- http://en.wikipedia.org/wiki/Jpg

* The Wikipedia articles are a bit dense; feel free to skim or skip!

diff pset5.pdf hacker5.pdf

- Hacker Edition challenges you to reduce (and enlarge) BMPs.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on Reddit or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Shorts

- Head to Week 7's shorts and watch the shorts on File I/O, Structs, and Valgrind. Just keep in mind that Jason's short on File I/O happens to focus on ASCII (i.e., text) files as opposed to binary files (like images). More on those later!
- You may also want to re-watch the short on GDB among Week 4's shorts!

Getting Started

- Welcome back!
- As always, first open a terminal window and execute

```
update50
```

  to make sure your appliance is up-to-date.
- Like Problem Set 4, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

```
cd ~/Dropbox
```

in order to navigate to your `~/Dropbox` directory. Then execute

```
wget http://cdn.cs50.net/2013/fall/psets/5/hacker5/hacker5.zip
```

in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

you should see that you now have a file called `hacker5.zip` in your `~/Dropbox` directory. Unzip it by executing the below.

```
unzip hacker5.zip
```

If you again execute

```
ls
```

you should see that you now also have a `hacker5` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f hacker5.zip
```

Now dive into that `hacker5` directory by executing the below.

```
cd hacker5
```

Now execute

```
ls
```

and you should see that the directory contains the below.

```
bmp/  jpg/  questions.txt
```

How fun! Two subdirectories and a file. Who knows what could be inside! Let's get started.

- If you ever saw Windows XP's default wallpaper (think rolling hills and blue skies), then you've seen a BMP. If you've ever looked at a webpage, you've probably seen a GIF. If you've ever looked at a digital photo, you've probably seen a JPEG. If you've ever taken a screenshot on a Mac, you've probably seen a PNG. Read up a bit on the BMP, GIF, JPEG,

and PNG file formats. Then, open up `questions.txt` in `~/Dropbox/hacker5`, as with `gedit`, and tell us the below.

1. How many different colors does each format support?
2. Which of these formats supports animation?
3. What's the difference between lossy and lossless compression?
4. Which of these formats is lossy-compressed?

- Curl up with the article from MIT at http://cdn.cs50.net/2013/fall/psets/5/garfinkel.pdf.

  Though somewhat technical, you should find the article's language quite accessible. Once you've read the article, answer each of the following questions in a sentence or more in `~/Dropbox/hacker5/questions.txt`.

  4. What happens, technically speaking, when a file is deleted on a FAT file system?
  5. What can someone like you do to ensure (with high probability) that files you delete cannot be recovered?

Whodunit and more

- Welcome to Tudor Mansion. Your host, Mr. John Boddy, has met an untimely end—he's the victim of foul play. To win this game, you must determine `whodunit`.

  Unfortunately for you (though even more unfortunately for Mr. Boddy), the only evidence you have is a 24-bit BMP file called `clue.bmp`, pictured below, that Mr. Boddy whipped up on his computer in his final moments. Hidden among this file's red "noise" is a drawing of `whodunit`.

  You long ago threw away that piece of red plastic from childhood that would solve this mystery for you, and so you must attack it as a computer scientist instead.

  But, first, some background.

- Perhaps the simplest way to represent an image is with a grid of pixels (i.e., dots), each of which can be of a different color. For black-and-white images, we thus need 1 bit per pixel, as 0 could represent black and 1 could represent white, as in the below. (Image adapted from http://www.brackeen.com/vga/bitmaps.html.)

  In this sense, then, is an image just a bitmap (i.e., a map of bits). For more colorful images, you simply need more bits per pixel. A file format (like GIF) that supports "8-bit color" uses 8 bits per pixel. A file format (like BMP, JPEG, or PNG) that supports "24-bit color" uses 24 bits per pixel. (BMP actually supports 1-, 4-, 8-, 16-, 24-, and 32-bit color.)

A 24-bit BMP like Mr. Boddy's uses 8 bits to signify the amount of red in a pixel's color, 8 bits to signify the amount of green in a pixel's color, and 8 bits to signify the amount of blue in a pixel's color. If you've ever heard of RGB color, well, there you have it: red, green, blue.

If the R, G, and B values of some pixel in a BMP are, say, 0xff, 0x00, and 0x00 in hexadecimal, that pixel is purely red, as 0xff (otherwise known as 255 in decimal) implies "a lot of red," while 0x00 and 0x00 imply "no green" and "no blue," respectively. Given how red Mr. Boddy's BMP is, it clearly has a lot of pixels with those RGB values. But it also has a few with other values.

Incidentally, HTML and CSS (languages in which webpages can be written) model colors in this same way. If curious, see http://en.wikipedia.org/wiki/Web_colors for more details.

Now let's get more technical. Recall that a file is just a sequence of bits, arranged in some fashion. A 24-bit BMP file, then, is essentially just a sequence of bits, (almost) every 24 of which happen to represent some pixel's color. But a BMP file also contains some "metadata," information like an image's height and width. That metadata is stored at the beginning of the file in the form of two data structures generally referred to as "headers" (not to be confused with C's header files). (Incidentally, these headers have evolved over time. This problem set only expects that you support version 4.0 (the latest) of Microsoft's BMP format, which debuted with Windows 95.) The first of these headers, called `BITMAPFILEHEADER`, is 14 bytes long. (Recall that 1 byte equals 8 bits.) The second of these headers, called `BITMAPINFOHEADER`, is 40 bytes long. Immediately following these headers is the actual bitmap: an array of bytes, triples of which represent a pixel's color. (In 1-, 4-, and 16-bit BMPs, but not 24- or 32-, there's an additional header right after `BITMAPINFOHEADER` called `RGBQUAD`, an array that defines "intensity values" for each of the colors in a device's palette.) However, BMP stores these triples backwards (i.e., as BGR), with 8 bits for blue, followed by 8 bits for green, followed by 8 bits for red. (Some BMPs also store the entire bitmap backwards, with an image's top row at the end of the BMP file. But we've stored this problem set's BMPs as described herein, with each bitmap's top row first and bottom row last.) In other words, were we to convert the 1-bit smiley above to a 24-bit smiley, substituting red for black, a 24-bit BMP would store this bitmap as follows, where `0000ff` signifies red and `ffffff` signifies white; we've highlighted in red all instances of `0000ff`.

```
ffffff  ffffff  0000ff  0000ff  0000ff  0000ff  ffffff  ffffff
ffffff  0000ff  ffffff  ffffff  ffffff  ffffff  0000ff  ffffff
0000ff  ffffff  0000ff  ffffff  ffffff  0000ff  ffffff  0000ff
0000ff  ffffff  ffffff  ffffff  ffffff  ffffff  ffffff  0000ff
0000ff  ffffff  0000ff  ffffff  ffffff  0000ff  ffffff  0000ff
0000ff  ffffff  ffffff  0000ff  0000ff  ffffff  ffffff  0000ff
```

```
ffffff  0000ff  ffffff  ffffff  ffffff  ffffff  0000ff  ffffff

ffffff  ffffff  0000ff  0000ff  0000ff  0000ff  ffffff  ffffff
```

Because we've presented these bits from left to right, top to bottom, in 8 columns, you can actually see the red smiley if you take a step back.

To be clear, recall that a hexadecimal digit represents 4 bits. Accordingly, `ffffff` in hexadecimal actually signifies `111111111111111111111111` in binary.

Okay, stop! Don't proceed further until you're sure you understand why `0000ff` represents a red pixel in a 24-bit BMP file.

- Okay, let's transition from theory to practice. Double-click **Home** on John Harvard's desktop and you should find yourself in John Harvard's home directory. Double-click **hacker5**, double-click **bmp**, and then double-click **smiley.bmp** therein. You should see a tiny smiley face that's only 8 pixels by 8 pixels. Select **View > Zoom > Zoom Fit**, and you should see a larger, albeit blurrier, version. (So much for "enhance," huh?) Actually, this particular image shouldn't really be blurry, even when enlarged. The program that launched when you double-clicked **smiley.bmp** (called Ristretto Image Viewer) is simply trying to be helpful (CSI-style) by "dithering" the image (i.e., by smoothing out its edges). Below's what the smiley looks like if you zoom in without dithering. At this zoom level, you can really see the image's pixels (as big squares).

  Okay, go ahead and return your attention to a terminal window, and navigate your way to `~/Dropbox/hacker5/bmp`. (Remember how?) Let's look at the underlying bytes that compose `smiley.bmp` using `xxd`, a command-line "hex editor." Execute:

  ```
  xxd -c 24 -g 3 -s 54 smiley.bmp
  ```

  You should see the below; we've again highlighted in red all instances of `0000ff`.

```
0000036: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
........................

000004e: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
........................

0000066: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
........................

000007e: 0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff
........................
```

```
0000096: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
.........................
00000ae: 0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff
.........................
00000c6: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
.........................
00000de: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
.........................
```

In the leftmost column above are addresses within the file or, equivalently, offsets from the file's first byte, all of them given in hex. Note that `00000036` in hexadecimal is `54` in decimal. You're thus looking at byte `54` onward of `smiley.bmp`. Recall that a 24-bit BMP's first 14 + 40 = 54 bytes are filled with metadata. If you really want to see that metadata in addition to the bitmap, execute the command below.

```
xxd -c 24 -g 3 smiley.bmp
```

If `smiley.bmp` actually contained ASCII characters, you'd see them in `xxd`'s rightmost column instead of all of those dots.

- So, `smiley.bmp` is 8 pixels wide by 8 pixels tall, and it's a 24-bit BMP (each of whose pixels is represented with 24 ÷ 8 = 3 bytes). Each row (aka "scanline") thus takes up (8 pixels) × (3 bytes per pixel) = 24 bytes, which happens to be a multiple of 4. It turns out that BMPs are stored a bit differently if the number of bytes in a scanline is not, in fact, a multiple of 4. In `small.bmp`, for instance, is another 24-bit BMP, a green box that's 3 pixels wide by 3 pixels wide. If you view it with Ristretto Image Viewer (as by double-clicking it), you'll see that it resembles the below, albeit much smaller. (Indeed, you might need to zoom in again to see it.)

Each scanline in `small.bmp` thus takes up (3 pixels) ×(3 bytes per pixel) = 9 bytes, which is not a multiple of 4. And so the scanline is "padded" with as many zeroes as it takes to extend the scanline's length to a multiple of 4. In other words, between 0 and 3 bytes of padding are needed for each scanline in a 24-bit BMP. (Understand why?) In the case of small.bmp, 3 bytes' worth of zeroes are needed, since (3 pixels) × (3 bytes per pixel) + (3 bytes of padding) = 12 bytes, which is indeed a multiple of 4.

To "see" this padding, go ahead and run the below.

```
xxd -c 12 -g 3 -s 54 small.bmp
```

Note that we're using a different value for `-c` than we did for `smiley.bmp` so that `xxd` outputs only 4 columns this time (3 for the green box and 1 for the padding). You should see output like the below; we've highlighted in green all instances of `00ff00`.

```
0000036: 00ff00 00ff00 00ff00 000000  ............
0000042: 00ff00 ffffff 00ff00 000000  ............
000004e: 00ff00 00ff00 00ff00 000000  ............
```

For contrast, let's use `xxd` on `large.bmp`, which looks identical to `small.bmp` but, at 12 pixels by 12 pixels, is four times as large. Go ahead and execute the below; you may need to widen your window to avoid wrapping.

```
xxd -c 36 -g 3 -s 54 large.bmp
```

You should see output like the below; we've again highlighted in green all instances of `00ff00`

```
0000036: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
000005a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
000007e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
00000a2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
00000c6: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
00000ea: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
000010e: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
0000132: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00
00ff00 00ff00 00ff00 00ff00  ....................................
```

```
0000156: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ................................

000017a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ................................

000019e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ................................

00001c2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00  ................................
```

Worthy of note is that this BMP lacks padding! After all, (12 pixels) × (3 bytes per pixel) = 36 bytes is indeed a multiple of 4.

Knowing all this has got to be useful!

- Okay, `xxd` only showed you the bytes in these BMPs. How do we actually get at them programmatically? Well, in `copy.c` is a program whose sole purpose in life is to create a copy of a BMP, piece by piece. Of course, you could just use `cp` for that. But `cp` isn't going to help Mr. Boddy. Let's hope that `copy.c` does!

  Go ahead and compile `copy.c` into a program called `copy` using `make`. (Remember how?) Then execute a command like the below.

```
./copy smiley.bmp copy.bmp
```

  If you then execute ls (with the appropriate switch), you should see that `smiley.bmp` and `copy.bmp` are indeed the same size. Let's double-check that they're actually the same! Execute the below.

```
diff smiley.bmp copy.bmp
```

  If that command tells you nothing, the files are indeed identical. (Note that some programs, like Photoshop, include trailing zeroes at the ends of some BMPs. Our version of `copy` throws those away, so don't be too worried if you try to copy a BMP that you've downloaded or made only to find that the copy is actually a few bytes smaller than the original.) Feel free to open both files in Ristretto Image Viewer (as by double-clicking each) to confirm as much visually. But diff does a byte-by-byte comparison, so its eye is probably sharper than yours!

  So how now did that copy get made? It turns out that `copy.c` relies on `bmp.h`. Let's take a look. Open up `bmp.h` (as with `gedit`), and you'll see actual definitions of those headers we've mentioned, adapted from Microsoft's own implementations thereof. In addition, that

file defines `BYTE`, `DWORD`, `LONG`, and `WORD`, data types normally found in the world of Win32 (i.e., Windows) programming. Notice how they're just aliases for primitives with which you are (hopefully) already familiar. It appears that `BITMAPFILEHEADER` and `BITMAPINFOHEADER` make use of these types. This file also defines a `struct` called `RGBTRIPLE` that, quite simply, "encapsulates" three bytes: one blue, one green, and one red (the order, recall, in which we expect to find RGB triples actually on disk).

Why are these `struct`s useful? Well, recall that a file is just a sequence of bytes (or, ultimately, bits) on disk. But those bytes are generally ordered in such a way that the first few represent something, the next few represent something else, and so on. "File formats" exist because the world has standardized what bytes mean what. Now, we could just read a file from disk into RAM as one big array of bytes. And we could just remember that the byte at location `[i]` represents one thing, while the byte at location `[j]` represents another. But why not give some of those bytes names so that we can retrieve them from memory more easily? That's precisely what the `struct`s in `bmp.h` allow us to do. Rather than think of some file as one long sequence of bytes, we can instead think of it as a sequence of `struct`s.

Recall that `smiley.bmp` is 8 by 8 pixels, and so it should take up $14 + 40 + 8 \cdot 8 \cdot 3 = 246$ bytes on disk. (Confirm as much if you'd like using `ls`.) Here's what it thus looks like on disk according to Microsoft:

As this figure suggests, order does matter when it comes to `struct`s' members. Byte `57 is `rgbtBlue` (and not, say, `rgbtRed`), because `rgbtBlue` is defined first in `RGBTRIPLE`. Our use, incidentally, of the `__attribute__` called `__packed__` ensures that `clang` does not try to "word-align" members (whereby the address of each member's first byte is a multiple of 4), lest we end up with "gaps" in our `struct`s that don't actually exist on disk.

Now go ahead and pull up the URLs to which `BITMAPFILEHEADER` and `BITMAPINFOHEADER` are attributed, per the comments in `bmp.h`. You're about to start using MSDN (Microsoft Developer Network)!

Rather than hold your hand further on a stroll through `copy.c`, we're instead going to ask you some questions and let you teach yourself how the code therein works. As always, `man` is your friend, and so, now, is MSDN. If not sure on first glance how to answer some question, do some quick research and figure it out! You might want to turn to http://www.cs50.net/resources/cppreference.com/stdio/ as well.

Allow us to suggest that you also run `copy` within `gdb` while answering these questions. Set a breakpoint at `main` and walk through the program. Recall that you can tell `gdb` to start running the program with a command like the below at `gdb`'s prompt.

```
run smiley.bmp copy.bmp
```

If you tell `gdb` to print the values of `bf` and `bi` (once read in from disk), you'll see output like the below, which we daresay you'll find quite useful.

```
{bfType = 19778, bfSize = 246, bfReserved1 = 0, bfReserved2 = 0,
  bfOffBits = 54}


{biSize = 40, biWidth = 8, biHeight = -8, biPlanes = 1, biBitCount =
24,
  biCompression = 0, biSizeImage = 192, biXPelsPerMeter = 2834,
  biYPelsPerMeter = 2834, biClrUsed = 0, biClrImportant = 0}
```

In `~/Dropbox/hacker5/questions.txt`, answer each of the following questions in a sentence or more.

6. What's `stdint.h`?
7. What's the point of using `uint8_t`, `uint32_t`, `int32_t`, and `uint16_t` in a program?
8. How many bytes is a `BYTE`, a `DWORD`, a `LONG`, and a `WORD`, respectively? (Assume a 32-bit architecture like the CS50 Appliance.)
9. What (in ASCII, decimal, or hexadecimal) must the first two bytes of any BMP file be? (Leading bytes used to identify file formats (with high probability) are generally called "magic numbers.)"
10. What's the difference between `bfSize` and `biSize`?
11. What does it mean if `biHeight` is negative?
12. What field in `BITMAPINFOHEADER` specifies the BMP's color depth (i.e., bits per pixel)?
13. Why might `fopen` return `NULL` in `copy.c:37`?
14. Why is the third argument to `fread` always `1` in our code?
15. What value does `copy.c:70` assign `padding` if `bi.biWidth` is `3`?
16. What does `fseek` do?
17. What is `SEEK_CUR`?

Okay, back to Mr. Boddy.

- Write a program called `whodunit` in a file called `whodunit.c` that reveals Mr. Boddy's drawing.

  Ummm, what?

  Well, think back to childhood when you held that piece of red plastic over similarly hidden messages. (If you remember no such piece of plastic, best to ask a classmate about his or her childhood.) Essentially, the plastic turned everything red but somehow revealed those messages. Implement that same idea in `whodunit`. Like `copy`, your program should accept exactly two command-line arguments. And if you execute a command like the below, stored in `verdict.bmp` should be a BMP in which Mr. Boddy's drawing is no longer covered with noise.

```
./whodunit clue.bmp verdict.bmp
```

  Allow us to suggest that you begin tackling this mystery by executing the command below.

```
cp copy.c whodunit.c
```

  Wink wink. You may be amazed by how few lines of code you actually need to write in order to help Mr. Boddy.

  There's nothing hidden in `smiley.bmp`, but feel free to test your program out on its pixels nonetheless, if only because that BMP is small and you can thus compare it and your own program's output with `xxd` during development. (Or maybe there is a message hidden in `smiley.bmp` too. No, there's not.)

  Rest assured that more than one solution is possible. So long as Mr. Boddy's drawing is identifiable (by you), no matter its legibility, Mr. Boddy will rest in peace.

  Because `whodunit` can be implemented in several ways, you won't be able to check your implementation's correctness with `check50`. And, lest it spoil your fun, the staff's solution to `whodunit` is not available.

- In `~/Dropbox/hacker5/questions.txt`, answer the question below. (And yet we used Photoshop.)

  18. Whodunit?

- Well that was fun. Bit late for Mr. Boddy, though.

  Let's have you write more than, what, two lines of code? Implement now in `resize.c` a program called `resize` that resizes 24-bit uncompressed BMPs by a factor of `f`. Your program should accept exactly three command-line arguments, per the below usage, whereby the first (`f`) must be a floating-point value in (0.0, 100.0], the second the name of the file to be resized, and the third the name of the resized version to be written.

```
Usage: resize f infile outfile
```

With a program like this, we could have created `large.bmp` out of `small.bmp` by resizing the latter by a factor of 4.0 (i.e., by multiplying both its width and it s height by 4.0), per the below.

```
./resize 4.0 small.bmp large.bmp
```

You're welcome to get started by copying (yet again) `copy.c` and naming the copy `resize.c`. But spend some time thinking about what it means to resize a BMP, particularly if `f` is in (0.0, 1.0). (You may assume that `f` times the size of `infile` will not exceed 2³² - 1. As for `f = 1.0`, the result should indeed be an `outfile` with dimensions identical to `infile`'s.) How you handle floating-point imprecision and rounding is entirely up to you, as is how you handle inevitable loss of detail. Decide which of the fields in BITMAPFILEHEADER and BITMAPINFOHEADER you might need to modify. Consider whether or not you'll need to add or subtract padding to scanlines.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/hacker5/resize bmp.h resize.c
```

And if you'd like to play with the staff's own implementation of `resize` in the appliance, you may execute the below.

```
~cs50/hacker5/resize
```

If you'd like to peek at, e.g., `large.bmp`'s headers (in a more user-friendly way than `xxd` allows), you may execute the below.

```
~cs50/hacker5/peek large.bmp
```

Better yet, if you'd like to compare your outfile's headers against the staff's, you might want to execute commands like the below while inside your `~/Dropbox/hacker5/bmp` directory. (Think about what each is doing.)

```
./resize 4 small.bmp student.bmp
~cs50/hacker5/resize 4 small.bmp staff.bmp
~cs50/hacker5/peek student.bmp staff.bmp
```

CSI (Computer Science Investigation)

- Alright, now let's put all your new skills to the test.

  In anticipation of this problem set, I spent the past several days snapping photos of people I know, all of which were saved by my digital camera as JPEGs on a 1GB CompactFlash (CF) card. (It's possible I actually spent the past several days on Facebook instead.) Unfortunately, I'm not very good with computers, and I somehow deleted them all! Thankfully, in the computer world, "deleted" tends not to mean "deleted" so much as "forgotten." My computer insists that the CF card is now blank, but I'm pretty sure it's lying to me.

  Write in `~/Dropbox/hacker5/jpg/recover.c` a program that recovers these photos.

  Ummm.

  Okay, here's the thing. Even though JPEGs are more complicated than BMPs, JPEGs have "signatures," patterns of bytes that distinguish them from other file formats. In fact, most JPEGs begin with one of two sequences of bytes. Specifically, the first four bytes of most JPEGs are either

```
0xff 0xd8 0xff 0xe0
```

  or

```
0xff 0xd8 0xff 0xe1
```

  from first byte to fourth byte, left to right. Odds are, if you find one of these patterns of bytes on a disk known to store photos (e.g., my CF card), they demark the start of a JPEG. (To be sure, you might encounter these patterns on some disk purely by chance, so data recovery isn't an exact science.)

  Fortunately, digital cameras tend to store photographs contiguously on CF cards, whereby each photo is stored immediately after the previously taken photo. Accordingly, the start of a JPEG usually demarks the end of another. However, digital cameras generally initialize CF cards with a FAT file system whose "block size" is 512 bytes (B). The implication is that these cameras only write to those cards in units of 512 B. A photo that's 1 MB (i.e., 1,048,576 B) thus takes up 1048576 ÷ 512 = 2048 "blocks" on a CF card. But so does a photo that's, say, one byte smaller (i.e., 1,048,575 B)! The wasted space on disk is called "slack space." Forensic investigators often look at slack space for remnants of suspicious data.

  The implication of all these details is that you, the investigator, can probably write a program that iterates over a copy of my CF card, looking for JPEGs' signatures. Each time you find a signature, you can open a new file for writing and start filling that file with bytes from my CF card, closing that file only once you encounter another signature. Moreover, rather than read my CF card's bytes one at a time, you can read 512 of them at a time into a buffer for efficiency's sake. Thanks to FAT, you can trust that JPEGs' signatures will be "block-aligned." That is, you need only look for those signatures in a block's first four bytes.

Realize, of course, that JPEGs can span contiguous blocks. Otherwise, no JPEG could be larger than 512 B. But the last byte of a JPEG might not fall at the very end of a block. Recall the possibility of slack space. But not to worry. Because this CF card was brand-new when I started snapping photos, odds are it'd been "zeroed" (i.e., filled with 0s) by the manufacturer, in which case any slack space will be filled with 0s. It's okay if those trailing 0s end up in the JPEGs you recover; they should still be viewable.

Now, I only have one CF card, but there are a whole lot of you! And so I've gone ahead and created a "forensic image" of the card, storing its contents, byte after byte, in a file called `card.raw`. So that you don't waste time iterating over millions of 0s unnecessarily, I've only imaged the first few megabytes of the CF card. But you should ultimately find that the image contains 50 JPEGs. As usual, you can open the file programmatically with fopen, as in the below. (It's fine to hard-code this path into your program rather than define it as some constant.)

```
FILE* file = fopen("card.raw", "r");
```

Notice, incidentally, that `~/Dropbox/hacker5/jpg` contains only `recover.c`, but it's devoid of any code. (We leave it to you to decide how to implement and compile `recover`!) For simplicity, you should hard-code `"card.raw"` in your program; your program need not accept any command-line arguments. When executed, though, your program should recover every one of the JPEGs from `card.raw`, storing each as a separate file in your current working directory. Your program should number the files it outputs by naming each `###.jpg`, where `###` is three-digit decimal number from `000` on up. (Befriend `sprintf`.) You need not try to recover the JPEGs' original names. To check whether the JPEGs your program spit out are correct, simply double-click and take a look! If each photo appears intact, your operation was likely a success!

Odds are, though, the JPEGs that the first draft of your code spits out won't be correct. (If you open them up and don't see anything, they're probably not correct!) Execute the command below to delete all JPEGs in your current working directory.

```
rm *.jpg
```

If you'd rather not be prompted to confirm each deletion, execute the command below instead.

```
rm -f *.jpg
```

Just be careful with that `-f` switch, as it "forces" deletion.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014/x/hacker5/recover recover.c
```

Lest it spoil your (forensic) fun, the staff's solution to `recover` is not available.

Sanity Checks

Before you consider this problem set done, best to ask yourself these questions and then go back and improve your code as needed! Do not consider the below an exhaustive list of expectations, though, just some helpful reminders. The checkboxes that have come before these represent the exhaustive list! To be clear, consider the questions below rhetorical. No need to answer them in writing for us, since all of your answers should be "yes!"

- Did you fill `questions.txt` with answers to all questions?
- Is the BMP that `whodunit` outputs legible (to you)?
- Does `resize` accept three and only three command-line arguments?
- Does `resize` ensure that `n` is in (0.0, 100.0]?
- Does `resize` update `bfSize`, `biHeight`, `biSizeImage`, and `biWidth` correctly?
- Does `resize` add or remove padding as needed?
- Does `recover` output 50 JPEGs? Are all 50 viewable?
- Does `recover` name the JPEGs `###.jpg`, where `###` is a three-digit number from `000` through `049`?
- Are all of your files where they should be in `~/Dropbox/hacker5`?