Table of Contents

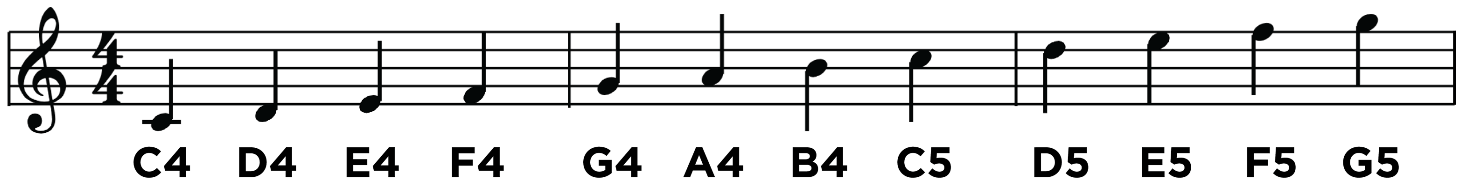
Music

tl;dr

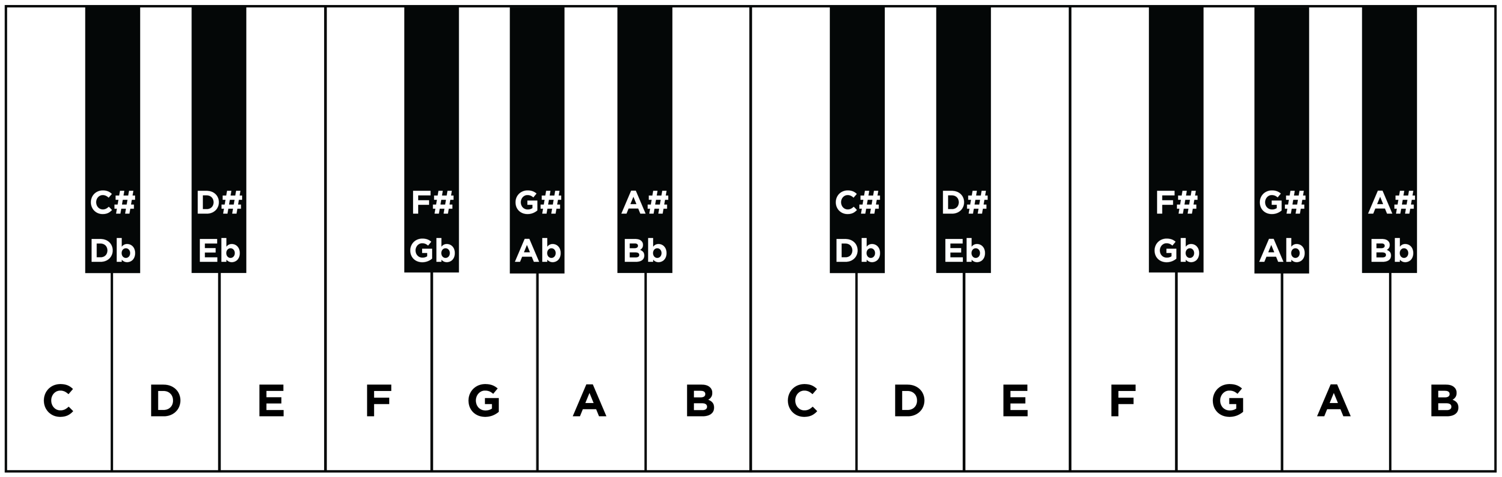
1. Learn to read sheet music.
2. Learn to read code.
3. Convert musical notes to frequencies.
4. Synthesize songs.

Background

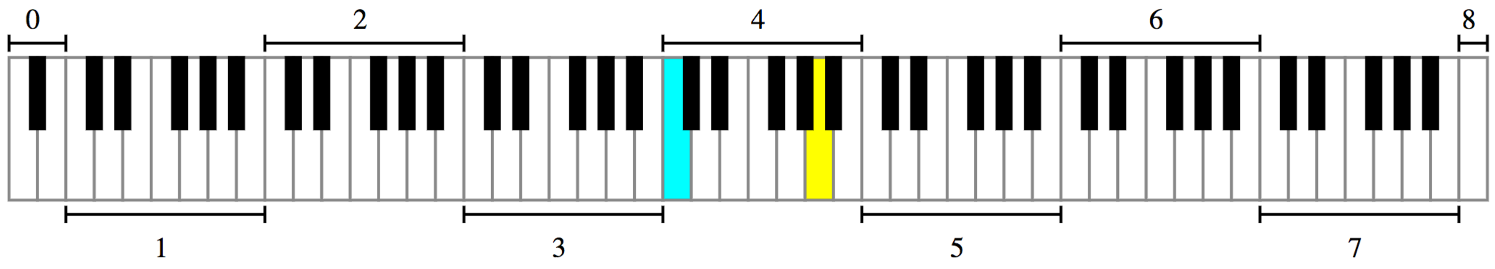
A song is essentially a sequence of sounds, otherwise known as notes, each of which has some duration. In Western music, each of those notes is known by a letter, A through G. Those letters happen to correspond to the white keys on a piano, otherwise known as "natural keys," per the below.



Among all of those white keys are black keys, each of which is identified by its proximity to a white key, per the below. A black key immediately above (i.e., to the right of) a white key is identified by the same letter but with a suffix of ♯ (often typed as `#`), otherwise known as a sharp; a black key immediately below (i.e., to the left of) a white key is also identified by the same letter but with a suffix of ♭ (often typed as `b`), otherwise known as a flat. Every key on a piano, meanwhile, is said to be one semitone, otherwise known as a half step, away from its adjacent neighbor, whether white or black. The effect of `#` and `b`, otherwise known as accidentals, is to raise or lower, respectively, the pitch of a note by one semitone.

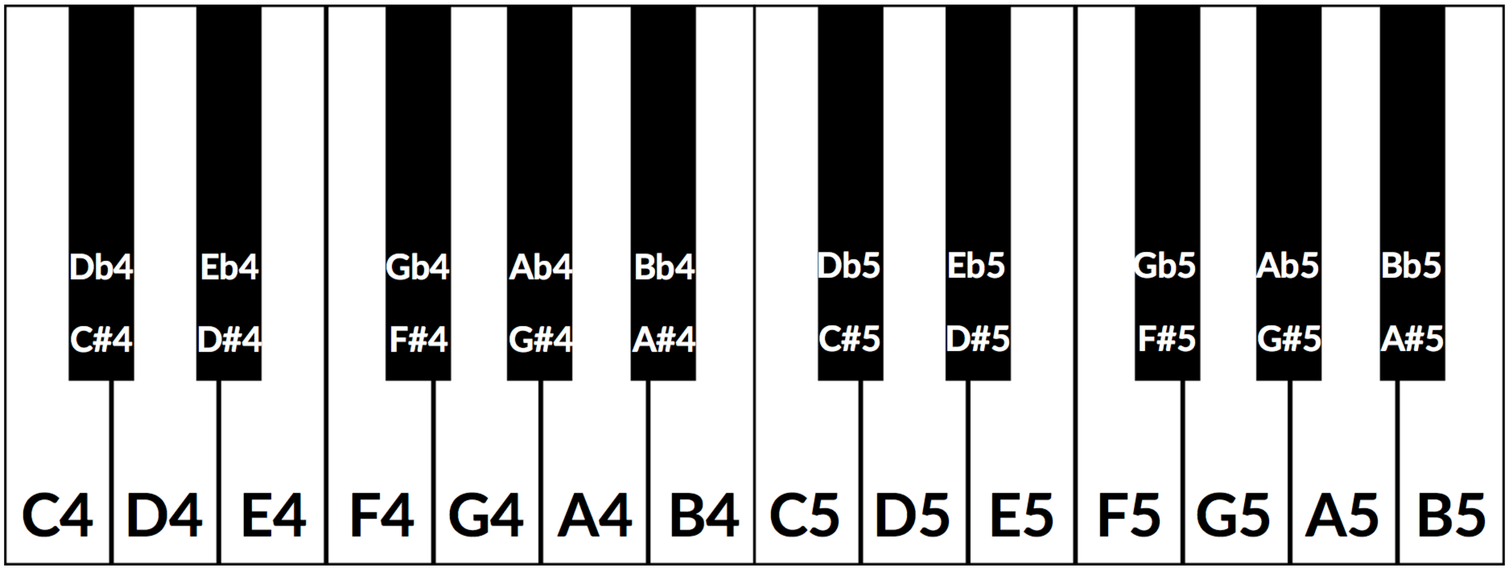


Pianos typically have as many as 88 keys, 52 of which are white. With only seven letters (A through G) with which to identify them, those letters necessarily identify multiple keys. And so notes are divided into octaves, groups of contiguous keys, each of which is numbered, per the below.

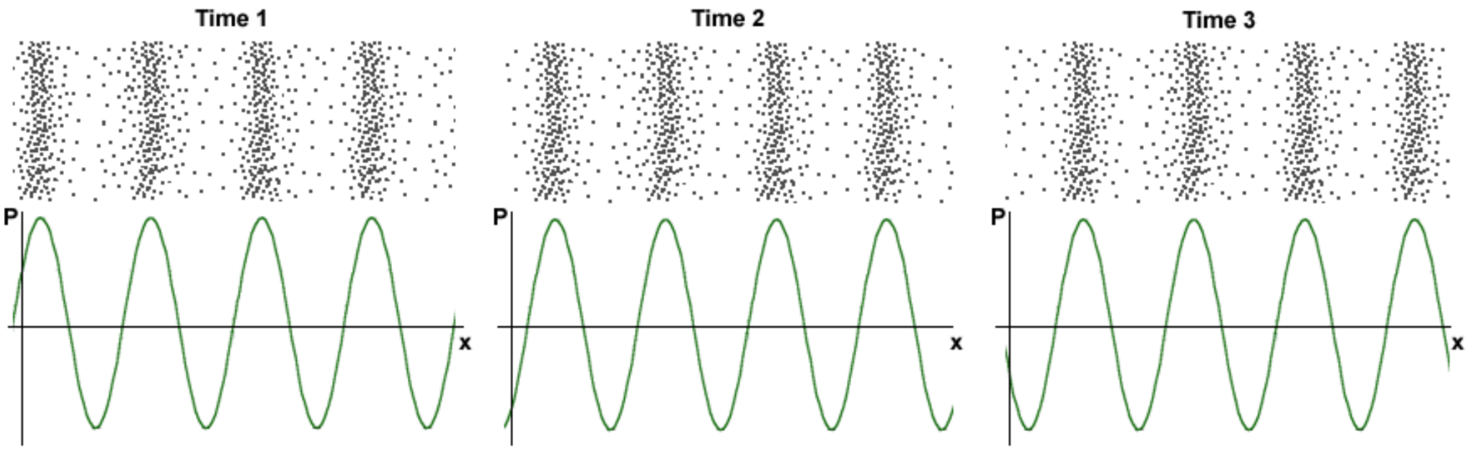


Source: https://en.wikipedia.org/wiki/A440_(pitch_standard) (https://en.wikipedia.org/wiki/A440_(pitch_standard))

Not only are notes identified by letters (and accidentals), then, but also by octaves, per the below.



Now, all of those keys, when pressed, generate vibrations and, in turn, waves of air molecules (i.e., alternations of high and low air pressure), otherwise known as sound waves, per the below. If those sound waves reach your ear, you'll hear sounds. Each of those sound waves travels at some rate, otherwise known as its frequency. The higher a sound wave's frequency, the higher the pitch of sound you'll hear; the lower a sound wave's frequency, the lower the pitch of sound you'll hear. If curious as to why some air molecules sound better than others, you might like the magical mathematics of music (https://plus.maths.org/content/magical-mathematics-music).



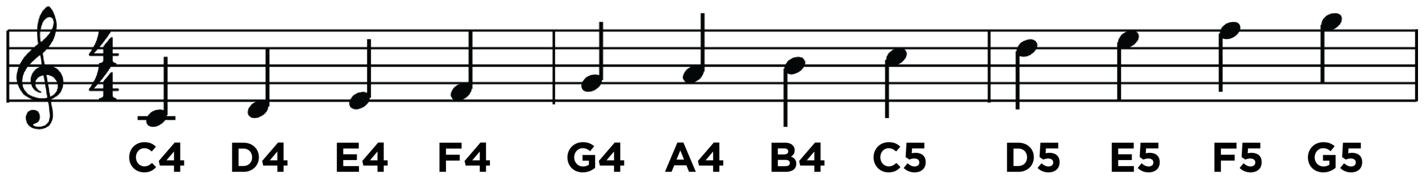
Chronological sequence of pictures of the compression of air molecules for a sound wave moving in the rightward direction. Source: https://web.stanford.edu/~zhoufan/MathematicsOfMusic.pdf (https://web.stanford.edu/~zhoufan/MathematicsOfMusic.pdf).

Among the most noteworthy (ha!) notes is Middle C, highlighted in cyan earlier, otherwise known as C4, since that C is in the piano's fourth octave. Above Middle C (i.e., to its right) is another notable (ha!) note, A4, highlighted in yellow earlier, otherwise known as A440, since the frequency of its sound waves is 440

Hz; which means that they oscilate up and down 440 times per second.

The frequencies of one octave’s notes differ from those of adjacent octaves' notes by a factor of two. For instance, the frequency of A3 is 220 Hz (i.e., half that of A4), while the frequency of A5 is 880 Hz (i.e., twice that of A4). More generally, the frequency, *f*, of some note is $2^{n/12} \times 440$, where *n* is the number of semitones from that note to A4, where *n* is negative if that note is below (i.e., to the left of) A4 and positive if that note is above (i.e., to the right of) A4.

Musicians, though, tend to write music not with letters or frequencies but with visual notations, otherwise known as sheet music, whereby notes are written on or between lines, otherwise known as a staff, with any accidentals positioned to the left of each note. The lines on or between which notes are written imply the notes' letters and octaves, per the below.



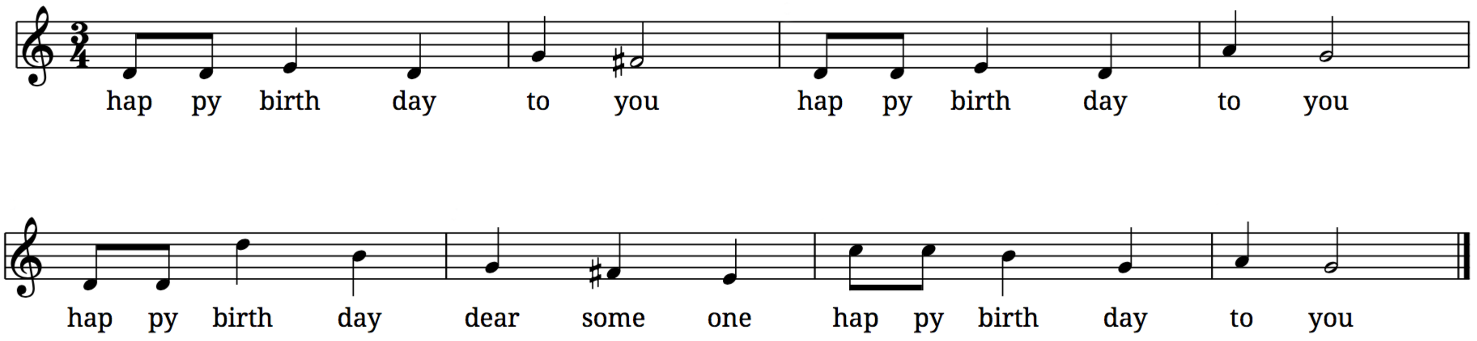
The duration of a note, meanwhile, is implied by its shape. For instance,

- ♪ is an eighth note, though when adjacent to one or more other eighth notes, they’re often beamed, a la ♫;
- ♩ is a quarter note, the duration of which is twice that of an eighth note;
- ♩. is a dotted quarter note, the duration of which is three times that of an eighth note;
- 𝅗𝅥 is a half note, the duration of which is four times that of an eighth note; and
- 𝅝 is a whole note, the duration of which is eight times that of an eighth note.

An absence of a note (i.e., silence) is considered a rest, the duration of which is also implied by its shape. For instance,

- 𝄾 is an eighth rest, the duration of which is identical to that of an eighth note;
- 𝄽 is a quarter rest, the duration of which is twice that of an eighth rest;
- 𝄼 is a half rest, the duration of which is four times that of an eighth rest; and
- 𝄻 is a whole rest, the duration of which is eight times that of an eighth rest.

With these building blocks can you represent a song like the below.


hap py birth day to you hap py birth day to you
hap py birth day dear some one hap py birth day to you

If unfamiliar, here's what that song sounds like (when the sound waves produced by its notes reach your ear).

0:00 / 0:12

Distribution

Included with this problem is a "distribution," some files that we've written that you'll first need to read (and understand!) before contributing improvements of your own. Unlike `cs50.h` and `stdio.h` and other header files you've been using for some time, which live somewhere in CS50 IDE, these files will live alongside your own code, where you can see them more easily.

Downloading

Here's how to download it.

First, execute

```
cd ~/workspace/pset3/
```

to ensure you're in `~/workspace/pset3/`. Then, execute

```
wget http://cdn.cs50.net/2017/fall/psets/3/music.zip
```

to download the distribution code as a ZIP (i.e., compressed file). If you then execute `ls`, you should see `music.zip` inside of your `pset3` directory. To unzip (i.e., uncompress) that file, execut

```
unzip music.zip
```

and then execute

```
rm music.zip
```

in order to delete the ZIP file itself. If you execute `ls`, you should now see a folder called `music` inside of your `pset3` directory. Then execute

```
cd music/
```

in order to change into that directory. And then execute `ls`. You should see the files and folder below, which collectively compose this problem's distribution!

```
Makefile  helpers.c  helpers.h  notes.c  synthesize.c  songs/  wav.c  wav.h
```

Understanding

Let's read through those files in order to understand them. Moving forward, reading (and understanding!) someone else's code, whether ours or some library's, will often be the first step in solving a problem. That way, you can build upon the work of others and solve even more interesting problems yourself!

`songs/`

First open up `songs/`, as with `cd` or CS50 IDE's file browser. In that directory are a bunch of `.txt` files, inside of which, it turns out, are a number of songs! Because ASCII alone doesn't lend itself to beautiful sheet music, we've instead adopted for these files a "machine-readable" format for songs instead. On each line of a file is a note and duration, separated by an `@`. For instance, atop `jeopardy.txt` (which you're welcome to open) are these lines:

```
G4@1/4
C5@1/4
G4@1/4
C4@1/4
G4@1/4
C5@1/4
G4@1/4
```

The first note in the theme song for Jeopardy is indeed a quarter note (per the `1/4`), specifically a G in the fourth octave. The second note is also a quarter note, but that one's a C in the fifth octave (a few keys to the right of the first one on a piano). Thereafter are five additional quarter notes.

Below those first seven lines in `jeopardy.txt`, notice, are two blank lines, the implication of which is that the seventh note is followed by two eighth rests (or, equivalently, one quarter rest). After those rests, the song resumes, resting only once more several notes later.

Make sense? Feel free to look through some of the other `.txt` files in `songs`. Cryptic though the files' lines might be at first glance, they're really just a top-down translation of (prettier) sheet music to a machine-readable text format, machine-readable in the sense that you're soon going to write code that reads those notes and durations!

`notes.c`

Next open up `notes.c`. In this file is a program (soon to be called `notes`) that not only prints the frequencies (in Hz) of all of the notes in an octave, it also outputs a WAV file (an audio file) via which you can hear those same notes. By default, it does so for the fourth octave, but if you pass it a command-line argument (a number between 0 and 8, inclusive), you can see and hear the frequencies of any octave's notes.

Read through the comments and code in `notes.c` and try to understand most, if not all, of its lines. Some might look unfamiliar. For instance, by convention, it uses a function called `fprintf` to print error messages to `stderr` (aka standard error) rather than `printf`, which, it turns out, prints to something called `stdout` (aka standard output). By default, messages printed to `stdout` and `stderr` both appear on the user's screens. But it's possible to separate them when running a program so that users can distinguish error messages from non-error messages. But more on that perhaps another time!

Notice, too, how `main` returns `1` in cases of error. That, too, is a convention. To date, we've not returned any values from `main`. But, recall that, all this time, `main` *has* had a return type, specifically `int`. It turns out, when `main` is done executing, it returns `0` by default, which, by convention, signifies success. If something goes wrong in a program, though, it's convention to return some value other than `0` (e.g., `1`). That value is called an "exit code" and can be used to distinguish one type of error from another. In fact, if you've ever seen a cryptic error code on your Mac's or PC's screen, it might very well have been the value returned by some (buggy) program's `main` function.

Notice too how this program uses a function called `sprintf` which doesn't actually print to the screen but instead stores its output in a string (hence the `s` in `sprintf`). We're using it in order to create a string from two placeholders, `%s` and `%i`. Notice how we allocate space for a (short) string by declaring an array for 4 `char`s. We then use `sprintf` to store a `NOTES[i]` (a `string`, ergo the `%s`) in that memory followed by `octave` (an `int`, ergo the `%i`). That way, we can take values like `"A"` and `4` and, effectively, concatenate them (i.e., append the latter to the former) in order to create a new `string`, the value of which is, for instance, `A4`.

Along the way in this program do we call some (presumably) unfamilar functions called `song_open`, `frequency`, `note_write`, and `song_close`. It turns out those functions are implemented in other files in this problem's distribution. Keep an eye out for them!

`synthesize.c`

In this file is a program (soon to be called `synthesize`) that synthesizes (i.e., generates) a song from a sequence of notes. Notice how it gets those notes from a user one at a time using `get_string`. It first checks, though, whether the user's input is a rest, as would happen if the user simply hits Enter. Else it proceeds to "tokenize" the user's input into two tokens: a note, which can be found to the left of the `@` in the user's input, and a fraction, which can be found to the right of the `@` in the user's input. The program uses a function called `strtok` to facilitate such. It then writes that note (or rest) to a file.

`wav.h`

Next open up `wav.h`, a header file used by both `notes.c` and `synthesize.c`. This file, together with `wav.c`, represents not a program but a "library," a set of functions that other programs can use as building blocks, much like `cs50` and `stdio` are libraries. This library's code just so happens to live in your work workspace now.

In `wav.h` too are definitions of two new data types, one called `note` and one called `song`. But more on those (and keywords like `typedef` and `struct` another time). For now, just notice how this file declares four functions (`note_write`, `rest_write`, `song_close`, and `song_open`), which `notes` and `synthesize` use.

`wav.c`

In `wav.c`, meanwhile, are the actual implementations of those functions plus a few others. Indeed, this file contains functions that implement support for WAV files, a popular (if dated) file format for audio. Those functions allow `notes` and `synthesize` to save notes to disk in files ending in `.wav`. To play those `.wav` files, simply open them via CS50 IDE's file browser. Or download them to your Mac or PC to play them locally.

No need to understand all of the code in `wav.c`, but you're welcome to read through it if you'd like!

`Makefile`

Next open up `Makefile`, the format of which is perhaps quite different from anything you've seen before. As its name might suggest, it's related to `make`, the program you've probably been using compile most of your programs, if only because compiling programs with `clang` itself tends to require more keystrokes. In previous problems, we've not needed a `Makefile`, which is essentially a configuration file for `make`, since `make` can infer how to compile a program that's composed of a single file (e.g., `hello.c`). But compiling both `notes` and `synthesize` requires multiple files, since both programs rely on `wav.h` and `wav.c`, plus two other files, `helpers.h` and `helpers.c`.

Simply executing

```
make notes
```

or

```
make synthesize
```

wouldn't provide nearly enough information for `make` to be able to infer which files it needs. So this `Makefile` exists so that `make` knows how to compile these programs.

`helpers.h`

In this file, now, are declarations for three functions:

- `duration`, which should take as input as a `string` a fraction (e.g., `1/4`) and return as an `int` a corresponding number of eigths (`2`, in this case, since `1/4` is equivalent to `2/8`);
- `frequency`, which should take as input as a `string` a note formatted as
  - `XY` (e.g., `A4`), where `X` is any of `A` through `G` and `Y` is any of `0` through `8`, or
  - `XYZ` (e.g., `A#4`), where `X` is any of `A` through `G`, `Y` is `#` or `b`, and `Z` is any of `0` through `8`,

  and return as an `int` the note's corresponding frequency, rounded to the nearest integer; and
- `is_rest`, which should return `true` if its input, a `string`, represents a rest in our machine-readable format, otherwise `false`.

`helpers.c`

And in this file there *should* be implementations of those three functions, but no! Not yet. That's where you come in!

Specification

`bday.txt`

In `bday.txt`, type the ASCII representation of *Happy Birthday*, translating its sheet music, above, to the machine-readable representation prescribed herein. You should find that the song begins with:

```
D4@1/8
D4@1/8
E4@1/4
D4@1/4
G4@1/4
F#4@1/2
```

`helpers.c`

`is_rest`

Complete the implementation of `is_rest` in `helpers.c`. Recall that blank lines represent rests in our machine-readable format. And recall that `synthesize` will call this function in order to determine if one of the lines a user has typed in is indeed blank.

What does it mean for a line to be blank? To answer that question, start by looking at `cs50.h` itself, wherein `get_string` is documented:

https://github.com/cs50/libcs50/blob/develop/src/cs50.h (https://github.com/cs50/libcs50/blob/develop/src/cs50.h)

What do the comments atop `get_string` say that the function returns if a user simply hits Enter, thereby inputting only a "line ending" (i.e., `\n`)?

When `is_rest` is subsequently passed such a `string`, `s`, how should it (nay, you!) recognize as much?

`duration`

Complete the implementation of `duration` in `helpers.c`. Recall that this function should take as input as a `string` a fraction and convert it into some integral number of eighths. You may assume that `duration` will only be passed a `string` formatted as `X/Y`, whereby each of `X` and `Y` is a positive decimal digit, and `Y` is, moreover, a power of 2.

`frequency`

Finally, complete the implementation of `frequency` in `helpers.c`. Recall that this function should take as input as a `string` a note (e.g., `A4`) and return its corresponding frequency in hertz as an `int`.

And recall that:

1. The frequency, *f*, of some note is $2^{n/12} \times 440$, where *n* is the number of semitones from that note to `A4`.
2. Each key on a piano is said to be one semitone, otherwise known as a half step, away from its adjacent neighbor, whether white or black.
3. The effect of `#` and `b`, otherwise known as accidentals, is to raise or lower, respectively, the pitch of a note by one semitone.

In implementing this function, you might find `pow` and `round`, both declared in `math.h`, of interest.

Walkthrough

Testing

To compile both `notes` and `synthesize`, execute

```
make
```

which should compile both at the same time, provided that `helpers.c` has no syntax errors.

To test your implementation of `frequency` in `helpers.c`, execute `notes`, which calls precisely that function, as via:

```
./notes
```

Confirm that the notes printed to the screen match your own calculations (whether on paper or calculator). You can also listen to the outputted `notes.wav` if you've an ear for the notes. Test other octaves by specifying them as command-line arguments, a la:

```
./notes 5
```

To test `frequency` further, along with `is_rest` and `duration`, execute `synthesize`, as via:

```
./synthesize test.wav
```

Then input one or more notes, one per line, and when done, hit ctrl-d to send `EOF` ("end of file") to `get_string` so that it breaks out of that program's loop. Open the resulting file (e.g., `test.wav`) by executing

```
open test.wav
```

or by double-clicking `test.wav` in CS50 IDE's file browser. Listen to the song to see (well, hear) if it sounds like (you think) it should!

Be sure to choose a different file name for each WAV file you synthesize, else your browser might cache (i.e., remember) and play an old version of a newly synthesized WAV file.

Typing notes into `synthesize`, though, will quickly become tedious. So you can instead leverage "input redirection" in order to pass whole files into `synthesize` as input. For instance, to pass all of the notes in `jeopardy.txt` into `synthesize` at once, execute:

```
./synthesize jeopardy.wav < songs/jeopardy.txt
```

Then execute

```
open jeopardy.wav
```

or simply double-click `jeopardy.wav` in CS50 IDE's file browser to open and (assuming no bugs!) listen to the song you just synthesized.

Correctness

```
check50 cs50/2018/x/music
```

Style

```
style50 helpers.c
```

Hints

As always, when writing code, take baby steps, only implementing enough lines to make progress before testing (and, if need be, debugging) your code. Only once that first step is succesful (i.e., debugged!) should you take another. Plan each of your steps by writing pseudocode before code.

In the context of `frequency` specifically, taking baby steps might mean:

1. Only implement support initially for `A0` through `A8`, no other notes. Ensure that `frequency` returns the expected values for those notes, as by running `notes` or using `debug50` or `eprintf`. Compare your function's output against your own calculations on paper or on a calculator.
2. Then add support for `#` and `b` but still only for `A0` through `A8` (i.e., `A#0` through `A#8` and `Ab0` through `Ab8`).
3. Then add support for `B`. Then for `C`. Then beyond.