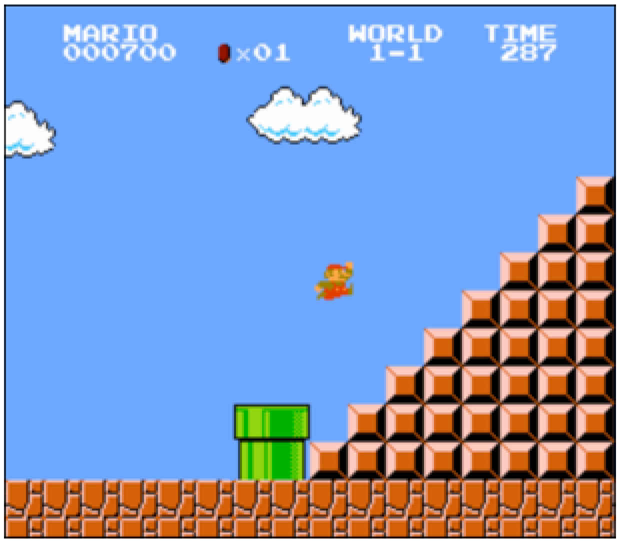
C Language Project Descriptions & Functionality

Mario

Toward the end of World 1-1 in Nintendo's Super Mario Brothers, Mario must ascend right-aligned pyramid of blocks, a la the below.



Let's recreate that pyramid in C, albeit in text, using hashes (`#`) for bricks, a la the below. Each hash is a bit taller than it is wide, so the pyramid itself is also be taller than it is wide.

Cash

When making change, odds are you want to minimize the number of coins you’re dispensing for each customer, lest you run out (or annoy the customer!). Fortunately, computer science has given cashiers everywhere ways to minimize numbers of coins due: greedy algorithms.

```
$ ./cash
Change owed: 0.41
4

$ ./cash
Change owed: -0.41
Change owed: foo
Change owed: 0.41
4
```

Readability

Implement a program that computes the approximate grade level needed to comprehend some text, per the below.

```
$ ./readability
Text: Congratulations! Today is your day. You're off to Great Places!
You're off and away!
Grade 3
```

Caesar

Implement a program that encrypts messages using Caesar's cipher, per the below.

```
$ ./caesar 13
plaintext:  HELLO
ciphertext: URYYB
```

Background

Supposedly, Caesar (yes, that Caesar) used to "encrypt" (i.e., conceal in a reversible way) confidential messages by shifting each letter therein by some number of places. For instance, he might write A as B, B as C, C as D, …, and, wrapping around alphabetically, Z as A. And so, to say HELLO to someone, Caesar might write IFMMP. Upon receiving such messages from Caesar, recipients would have to "decrypt" them by shifting letters in the opposite direction by the same number of places.

The secrecy of this "cryptosystem" relied on only Caesar and the recipients knowing a secret, the number of places by which Caesar had shifted his letters (e.g., 1). Not particularly secure by modern standards, but, hey, if you're perhaps the first in the world to do it, pretty secure!

Unencrypted text is generally called *plaintext*. Encrypted text is generally called *ciphertext*. And the secret used is called a *key*.

To be clear, then, here's how encrypting `HELLO` with a key of 1 yields `IFMMP`:

plaintext	H	E	L	L	O
+ key	1	1	1	1	1
= ciphertext	I	F	M	M	P

Plurality

Implement a program that runs a plurality election, per the below.

```
$ ./plurality Alice Bob Charlie
Number of voters: 4
Vote: Alice
Vote: Bob
Vote: Charlie
Vote: Alice
Alice
```

Background

Elections come in all shapes and sizes. In the UK, the Prime Minister is officially appointed by the monarch, who generally chooses the leader of the political party that wins the most seats in the House of Commons. The United States uses a multi-step Electoral College process where citizens vote on how each state should allocate Electors who then elect the President.

Perhaps the simplest way to hold an election, though, is via a method commonly known as the “plurality vote” (also known as “first-past-the-post” or “winner take all”). In the plurality vote, every voter gets to vote for one candidate. At the end of the election, whichever candidate has the greatest number of votes is declared the winner of the election.

Runoff

Implement a program that runs a runoff election, per the below.

```
./runoff Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Bob
Rank 3: Charlie

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Alice
Rank 3: Charlie

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Alice
```

Filter

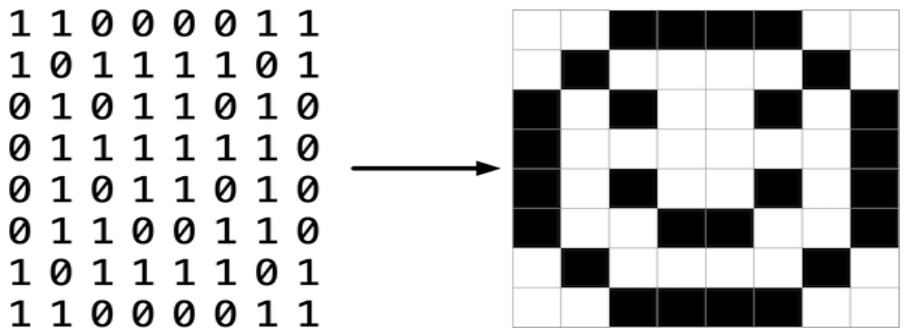
Implement a program that applies filters to BMPs, per the below.

```
$ ./filter -r image.bmp reflected.bmp
```

Background

Bitmaps

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.



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 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 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.

Recover

Implement a program that recovers JPEGs from a forensic image, per the below.

```
$ ./recover card.raw
```

Background

In anticipation of this problem, we spent the past several days taking photos of people we know, all of which were saved on a digital camera as JPEGs on a memory card. (Okay, it's possible we actually spent the past several days on Facebook instead.) Unfortunately, we somehow deleted them all! Thankfully, in the computer world, "deleted" tends not to mean "deleted" so much as "forgotten." Even though the camera insists that the card is now blank, we're pretty sure that's not quite true. Indeed, we're hoping (er, expecting!) you can write a program that recovers the photos for us!

Speller

Be sure to read this specification in its entirety before starting so you know what to do and how to do it!

Implement a program that spell-checks a file, a la the below, using a hash table.

```
$ ./speller texts/lalaland.txt
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

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