This is CS50

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Bulbs

Not-So-Broken Light Bulbs

In lecture, you may have noticed what seemed like a "bug" at the front of the stage, whereby some of the bulbs always seem to be off:

Each sequence of bulbs, though, encodes a message in *binary*, the language computers “speak.” Let’s write a program to make secret messages of your own, perhaps that we could even put on stage!

Getting Started

Open VS Code (https://cs50.dev/).

Start by clicking inside your terminal window, then execute `cd` by itself. You should find that its “prompt” resembles the below.

```
$
```

Click inside of that terminal window and then execute

```
wget https://cdn.cs50.net/2022/fall/psets/2/bulbs.zip
```

followed by Enter in order to download a ZIP called `bulbs.zip` in your codespace. Take care not to overlook the space between `wget` and the following URL, or any other character for that matter!

Now execute

```
unzip bulbs.zip
```

to create a folder called `bulbs`. You no longer need the ZIP file, so you can execute

```
rm bulbs.zip
```

and respond with “y” followed by Enter at the prompt to remove the ZIP file you downloaded.

Now type

```
cd bulbs
```

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

```
bulbs/ $
```

If all was successful, you should execute

```
ls
```

and see a file named `bulbs.c`. Executing `code bulbs.c` should open the file where you will type your code for this problem set. If not, retrace your steps and see if you can determine where you went wrong!

Implementation Details

To write our program, we’ll first need to think about **bases**.

The Basics

The simplest *base* is base-1, or *unary*; to write a number, *N*, in base-1, we would simply write *N* consecutive `1`s. So the number `4` in base-1 would be written as `1111`, and the number `12` as `111111111111`. Think of it like counting on your fingers or tallying up a score with marks on a board.

You might see why base-1 isn’t used much nowadays. (The numbers get rather long!) Instead, a common convention is base-10, or *decimal*. In base-10, each *digit* is multiplied by some power of 10, in order to represent larger numbers. For instance, 123 is short for $123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$.

Changing base is as simple as changing the 10 above to a different number. For instance, if you wrote `123` in base-4, the number you’d really be writing is $123 = 1 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0$, which is equal to the decimal number 27.

Computers, though, use base-2, or *binary*. In binary, writing `123` would be a mistake, since binary numbers can only have `0`s and `1`s. But the process of figuring out exactly what decimal number a binary number stands for is exactly the same. For instance, the number `10101` in base-2 represents $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, which is equal to the decimal number 21.

Encoding a Message

Light bulbs can only be on or off. In other words, light bulbs represent two possible states; either the bulb is on, or the bulb is off, just as binary numbers are either 1 or 0. We'll have to find a way to encode text as a sequence of binary numbers.

Let's write a program called `bulbs` that takes a message and converts it to a set of bulbs that we could show to an unsuspecting audience. We'll do it in two steps:

- The first step consists of turning the text into decimal numbers. Let's say we want to encode the message `HI!`. Luckily, we already have a convention in place for how to do this, ASCII (https://asciitable.com/). Notice that `H` is represented by the decimal number `72`, `I` is represented by `73`, and `!` is represented by `33`.
- The next step involves taking our decimal numbers (like `72`, `73`, and `33`) and converting them into equivalent binary numbers, which only use 0s and 1s. For the sake of having a consistent number of bits in each of our binary numbers, assume that each decimal is represented with 8 bits. `72` is `01001000`, `73` is `01001001`, and `33` is `00100001`.

Lastly, we'll interpret these binary numbers as instructions for the light bulbs on stage; 0 is off, 1 is on. (You'll find that `bulbs.c` includes a `print_bulb` function that's been implemented for you, which takes in a `0` or `1` and outputs emoji representing light bulbs.)

Here's an example of how the completed program might work. Unlike the Sanders stage, we'll print one byte per line for clarity.

```
# ./bulbs
Message: HI!
⚫⚪⚫⚫⚪⚫⚫⚫
⚫⚪⚫⚫⚪⚫⚫⚪
⚫⚫⚪⚫⚫⚫⚫⚪
```

To check our work, we can read a bulb that's on (⚪) as a `1` and bulb that's off (⚫) as a `0`. Then `HI!` became

```
01001000
01001001
00100001
```

which is precisely what we'd expect.

Another example:

```
# ./bulbs
Message: HI MOM
⚫🟡⚫⚫🟡⚫⚫⚫
⚫🟡⚫⚫🟡⚫⚫🟡
⚫⚫🟡⚫⚫⚫⚫⚫
⚫🟡⚫⚫🟡🟡⚫🟡
⚫🟡⚫⚫🟡🟡🟡🟡
⚫🟡⚫⚫🟡🟡⚫🟡
```

Notice that all characters are included in the lightbulb instructions, including nonalphabetical characters like spaces (`00100000`).

Specification

Design and implement a program, `bulbs`, that converts text into instructions for the strip of bulbs on CS50's stage as follows:

- Implement your program in a file called `bulbs.c`.
- Your program must first ask the user for a message using `get_string`.
- Your program must then convert the given `string` into a series of 8-bit binary numbers, one for each character of the string.
- You can use the provided `print_bulb` function to print a series of `0`s and `1`s as a series of yellow and black emoji, which represent on and off light bulbs.
- Each "byte" of 8 symbols should be printed on its own line when outputted; there should be a `\n` after the last "byte" of 8 symbols as well.

▶ **Hints for Decimal-to-Binary**

How to Test Your Code

Your program should behave per the examples above. You can check your code using `check50`, a program that CS50 will use to test your code when you submit, by typing in the following at the `$` prompt. But be sure to test it yourself as well!

```
check50 cs50/problems/2023/x/bulbs
```

To evaluate that the style of your code, type in the following at the `$` prompt.

```
style50 bulbs.c
```

How to Submit

In your terminal, execute the below to submit your work.

```
submit50 cs50/problems/2023/x/bulbs
```