Assignment 3
Hangman

by Jess Srinivas and Ben Grant
edits by Kerry Veenstra
CSE 13S, Spring 2024
Document version 1 (changes in Section 4)

Due Wednesday, May 1st, 2024, at 11:59 pm
Draft Due Monday, April 29th, 2024, at 11:59 pm

1 Introduction

Many of you have played the classic word game, Hangman (`https://en.wikipedia.org/wiki/Hangman_(game)`). Despite its morbid themes, hangman is sometimes played as a tool to teach vocabulary. In this assignment, you will be implementing the game in C.

2 Strings and Arrays

Hangman is a word game, and to create a word game, you must first have a representation for a word. You may recall that a `char` in C is actually just a signed 8-bit integer[1]. To use that integer as a character, we use an ASCII table (which is only defined for positive numbers). A string is just an array of characters. To allow you to complete this assignment, we need to explain all of these in depth, and provide you tools to work with them.

2.1 Arrays

Arrays and memory are a topic that we will cover in a lot greater detail soon in this class, but we will cover the basics. Arrays in C work by storing data in sequential order in memory. When you write a line like, `int arr[3];` the space that the data is stored in is set aside when the program is compiled, meaning that you can't increase the size of an array.

Arrays can also implicitly convert to pointers. In this case, you will end up with a pointer to the first element of the array. However, it's still important to keep in mind that arrays and pointers *are not the same thing*. One thing that demonstrates the difference is the `sizeof` operator, which tells you how much space a variable takes up in bytes. `sizeof` an array is the size of each element multiplied by the number of elements, whereas `sizeof` a pointer is a constant for each CPU architecture (8 on modern computers that use 64-bit processors).

2.2 Strings

Strings in C are just arrays of characters, followed by a null byte (a character whose decimal value is 0). The null byte is widely used by library functions to indicate the length of the string, so omitting a null terminator will likely lead to errors. Fortunately, the compiler automatically inserts null bytes for string literals. You can index into and iterate over a string just like an array. There is also a library for interacting with strings, `<string.h>`. For more information, see the `man` pages on this library.

[1] This is the case for your virtual machine, but may be different on other computers: they are not guaranteed to be signed or 8 bits. The important part is that you can treat them as numbers.

3 Your task

3.1 Workflow

1. Complete and submit your design doc by Monday, April 29th.
2. Implement and test the functions in the template file, `hangman_helpers.c`
3. Complete your `main` function in `hangman.c` and test your code.
4. Fix any issues with your design draft, and finish the results section.
5. Submit your final commit ID by Wednesday, May 1st.

3.2 Starter Files

The files we will provide to you to help you with this are as follows:

- `Makefile`: To help you build and run your code
- `design_template.zip`: For you to upload to `https://overleaf.com` and modify to make your draft and final report.
- `hangman_helpers.h`: This is a header file with a few strings defined for consistency in grading, and a few function definitions that you need to fill out.
- `hangman_helpers.c`: This file is a template for how you should fill out all the rest of your functions.

3.3 Required functions

These functions must appear in `hangman_helpers.c`

`bool string_contains_character(const char *s, char c)`

Checks if the string `s` contains the character `c`. If it does, it returns `true`, otherwise, it returns `false`. You may assume that `s` is a properly null terminated string, less or equal to than 256 characters.

`char read_letter(void)`

Prompts the user for a guess and reads in one letter (or any other character) from `stdin`. It then returns it. You may assume that the user only inputs one character followed by a newline.

`bool is_lowercase_letter(char c)`

Checks if the character `c` is a lowercase letter. If it is not, it returns false. You may not use any functions from `ctype.h` in this function, or anywhere else in your code.

`bool is_valid_secret(const char *secret)`

Takes in a string called `secret` and checks if it is a valid hangman secret. If it is not, it will print the first invalid character in the form `invalid character: 'X'`, where `X` is the character that is invalid and return false. If it is, it prints nothing and returns true. The error message must print to `stdout`. If the secret is too long, it will print `the secret phrase is over 256 characters`. The secret can be exactly 256 characters.

Recall that a valid secret word is all lowercase, but may include spaces, apostrophes, and hyphens.

The argument here is `const` so we can't modify the secret on accident when we check if its valid.

3.4 Assignment specifics

For this assignment, the hangman game that you create will have a few modifications. For one, the secret can be multiple words, and can include apostrophes, spaces, and hyphens. A secret must be provided, and at most 256. If these special characters are present, they will be shown at the start of the game. If only special characters are present, the player has already won. The player will guess only one letter at a time. You do not need to handle errors if they chose to show more than that. If the user enters an invalid secret, you should print the exact error messages that are printed in the resources binary, and exit with any non-zero exit code.

We have provided you with a few strings in `hangman_helpers.h`. There is also an array of "ASCII art" for hangman. You must also print these exactly.

In the beginning of the game, you will print an empty gallows, a line for the phrase, and a list of eliminated letters. You will also print a new line and a prompt to have the player guess a letter.

```
  _____
 |     |
       |
       |
       |\
       | \
     __|__\

    Phrase: ___'_ __ __ _____-______
Eliminated:

Guess a letter:
```

In this example, the phrase is `don't go in empty-handed`. Note how the colons after "Phrase" and "Eliminated" align. Your output must match ours, and this is a good start to doing so.

The game will then prompt the player repeatedly to enter a character, and update the gallows, phrase, and eliminated sections. Any error handling for this section should closely resemble the binary provided to you.

If the player loses, you will print the following

```
Guess a letter: b
  _____
 |     |
(_)    |
/|\    |
 |     |\
/ \    | \
     __|__\

    Phrase: d__'_ g_ i_ e____-h__ded
Eliminated: bcfjkl

You lose! The secret phrase was: don't go in empty-handed
```

In this example, the player guessed these letters: l, k, j, i, h, g, f, e, d, c, and b (that is, reverse alphabetical order, starting at l). Note that the *eliminated letters print in alphabetical order*. This is how the eliminated letters should always print.

If a player guesses any letter that they have already guessed, we should prompt the player for another letter without reprinting the gallows and phrase. Here someone guesses `a` and then `b` and then guesses `a` and `b` again:

```
  _____
 |     |
(_)    |
       |
       |\
       | \
     __|__\

    Phrase: ___'_ __ __ _____-_a____
Eliminated: b

Guess a letter: a
Guess a letter: b
Guess a letter:
```

If the player wins, you will reprint the gallows and phrase, along with a message telling the player they won.

```
Guess a letter: a
  _____
 |     |
       |
       |
       |\
       | \
     __|__\

    Phrase: don't go in empty-handed
Eliminated:

You win! The secret phrase was: don't go in empty-handed
```

In this case, the player managed to guess the entire phrase without making any mistakes.

We highly recommend (preferably automated) testing using the example binary provided as a reference. We also recommend that you write testing code to make sure that the code in your `hangman_helpers.c` file works as expected. You should know how to do both of these. If you do not, please see a tutor or TA, and they will help you with the process of testing.

3.5 Testing

The `resources` repository has three files that you can use for testing. Use `tester.txt` as input for specific command lines (below). Then compare the output of your `./hangman` program with the resource files `expected_win.txt` and `expected_lose.txt`.

Here is a test command. Notice that the command-line argument to `./hangman` is the letters of the alphabet in reverse order. Compare the result (`win.txt`) with `expected_win.txt`. (Remember `diff`?)

```
./hangman "zyxwvutsrqponmlkjihgfedcba" < tester.txt > win.txt
```

Here is another test command. The command-line argument to `./hangman` is the phrase `"don't go in empty-handed"`. Compare the result (`lose.txt`) with `expected_lose.txt`.

```
./hangman "don't go in empty-handed" < tester.txt > lose.txt
```

Can you write test scripts that run these two commands?

3.6 Submission

These are the files we require from you:

- `Makefile`: To help you build and run your code. This should build all targets specified in the template with the following compiler flags: `-Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic`
- `design.pdf`: Instructions can be found in the template.
- `hangman_helpers.h`: This is a header file with a few strings defined for consistency in grading, and a few function definitions that you will fill out in `hangman_helpers.c`. You may add functions to this file, but may not modify anything that is currently there.
- `hangman_helpers.c`: This file is a template for how you should fill out all the rest of your functions. You may NOT use any functions from `ctype.h` in this file.
- `hangman.c`: This should contain anything a main function that you use to play hangman. You must use all four of the functions provided in `hangman_helpers.h`

4 Revisions

Version 1 Original.