CSC 230 Project 3

Hangman Game

For this project, you're going to write a program that allows the user to play a simple hangman game. In the game of hangman, a player tries to guess a word, which is initially displayed with each letter replaced by an underscore. The player guesses the word by continually choosing letters of the alphabet that may or may not be in the word. If the letter *is in the word*, each underscore that corresponds to the chosen letter is replaced by the letter. If the letter *is not in the word*, a body part is added to a stick figure, which is typically hanging from a gallows, hence the name `hangman`. If the player guesses the word before the stick figure is complete, the player wins. But if the figure is completed before the word is guessed, the player loses and the word is revealed. In our game, the stick figure will consist of 7 body parts so if the player guesses 7 letters that are not in the word, the player loses.

The following shows a player's interaction with the game using the provided animals.txt file and a seed of 2, which is used for the random number generator (more about this later). The player wins the first time, loses the second time, and then opts not to play again.

```
$ ./hangman animals.txt 2

_ _ _

Remaining letters: a b c d e f g h i j k l m n o p q r s t u v w x y z

letter> a

_ a _

Remaining letters: b c d e f g h i j k l m n o p q r s t u v w x y z

letter> z

 O

_ a _

Remaining letters: b c d e f g h i j k l m n o p q r s t u v w x y

letter> tag

Invalid letter

letter> 12

Invalid letter
```

```
letter> a

Invalid letter

letter> t

 O

_ a t

Remaining letters: b c d e f g h i j k l m n o p q r s u v w x y

letter> c

 O
 |

_ a t

Remaining letters: b d e f g h i j k l m n o p q r s u v w x y

letter> y

 O
 |
 |

_ a t

Remaining letters: b d e f g h i j k l m n o p q r s u v w x

letter> b

b a t

You win!

Play again(y,n)> yes

_ _ _ _

Remaining letters: a b c d e f g h i j k l m n o p q r s t u v w x y z

letter> a

 O

_ _ _ _

Remaining letters: b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
letter> o

 O
 |


_ _ _ _

Remaining letters: b c d e f g h i j k l m n p q r s t u v w x y z

letter> c

 O
 |
 |


_ _ _ _

Remaining letters: b d e f g h i j k l m n p q r s t u v w x y z

letter> d

 O
 |
 |

d _ _ _

Remaining letters: b e f g h i j k l m n p q r s t u v w x y z

letter> x

 O
/|
 |

d _ _ _

Remaining letters: b e f g h i j k l m n p q r s t u v w y z

letter> y

 O
/|\
 |

d _ _ _

Remaining letters: b e f g h i j k l m n p q r s t u v w z

letter> z
```

```
 O
/|\
 |
/

d _ _ _

Remaining letters: b e f g h i j k l m n p q r s t u v w

letter> g

 O
/|\
 |
/ \

You lose!
Word was deer

Play again(y,n)> goodbye
```

As in project 2, you'll be developing this project using git for revision control. You should be able to just unpack the starter into the p3 directory of your cloned repo to get started. See the Getting Started section for instructions.

This homework supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 3

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow.

Requirements

You're going to write a program, `hangman`, that allows the user to try to guess one or more words.

At start-up, the program will read a list of words and optionally a seed for the random number generator used to select a word to guess, which allows for repeatable testing of the game. The word with each letter replaced by an underscore will then be displayed. The user will continually enter a letter that may or may not be in the word. If the letter is in the word, the word will be redisplayed with the letter replacing the corresponding underscore(s). If the

letter is not in the word, a body part will be added to the stick figure. If the user chooses a letter that was previously chosen, a character that is not a valid letter, or anything other than a single lowercase letter, an error message will be output and the user will be reprompted to enter a letter.

After winning or losing, the user is asked if they would like to play again. If they enter anything that starts with y or Y, they are given another word to guess. Otherwise, the program ends.

Command-Line Arguments and Word List

The `hangman` program will take one required command-line argument, the name of a file containing the word list, and one optional command-line argument, an integer seed used for the random number generator to enable repeatable testing.

The word list is a list of words that the user will try to guess. The file should contain a whitespace-separated list of words, with each word consisting of lower-case letters and no word being longer than 20 letters. The file may contain at most 50 words.

If the user gives the wrong number of command line arguments or a seed that is not a positive integer, your program should print the following usage message to standard error and terminate with an exit status of 1.

```
usage: hangman <word-file> [seed]
```

If the program can't open the given word file, it should print the following message to standard error and exit with a status of 1.

```
Can't open word file
```

If anything is wrong with the contents of the word file (e.g., too many words, words that are too long, words that contain non-lowercase letters), it should print the following message to standard error and exit with a status of 1.

```
Invalid word file
```

Game Play and User Input

The word to be guessed will initially be displayed with each letter replaced by an underscore. The underscores are separated by a single space with no space after the final underscore. Each time the user guesses a letter contained in the word, each occurrence of letter will be displayed instead of an underscore. Each time the user guesses a letter that is not contained in the word, a body part is added to the stick figure in this order: head, torso top, torso bottom, left arm (on *your left*, technically, the figure's right arm), right arm, left leg, right leg. The stick figure with all 7 body parts is shown below:

```
 o
/|\
 |
```

```
/ \
```

The program will repeatedly display the letters that have not been guessed, separated by a single space with no space after the final letter, and then prompt the user for a letter with the following prompt (there's a space at the end, after the >):

```
letter>
```

The user must enter exactly one lowercase letter.

The program will respond by displaying the stick figure parts, if any, followed by the word in which any correctly guessed letters are displayed and an underscore represents any letter that has not been guessed.

For example, imagine the word being guessed is "bat", and the user has guessed the letters 'a', 'z', 't', and 'c'. The program should output the stick figure with 2 body parts due to the 2 incorrectly guessed letters, the word with an underscore followed by the 2 correctly guessed letters, and the 22 letters that have not yet been guessed, before prompting the user for the next letter:

```
 O
 |

_ a t

Remaining letters: b d e f g h i j k l m n o p q r s u v w x y

letter>
```

If the user enters invalid input (e.g., too long, containing invalid characters, a letter that has been previously guessed), the program will print the following message to standard output, ignore that input line, and then prompt the user for another letter.

```
Invalid letter
```

The game is over when each letter has been guessed or all 7 body parts have been added to the figure. If the word was guessed, the following message will be output.

```
You win!
```

If the word was not guessed, the message below will be output followed by the word, for example,

```
You lose!
Word was deer
```

In either case, the user will be asked if they want to play again as shown below.

Play again(y,n)>

If they enter anything that starts with 'y' or 'Y', they will be given another word to guess. Otherwise, the program will terminate successfully. The program will also terminate successfully if it reaches end-of-file on standard input (remember, if you're running the program interactively, you can type CTRL-D on Unix or CTRL-Z on windows to indicate end-of-file).

Formatting

Please see the examples and the sample input and expected output files provided with the starter for the specifics on how the output is formatted in terms of blank lines, etc.

Design

The `hangman` program will be defined in three components:

- `wordlist.c` and `wordlist.h`
  This component defines functions for working with the word list.

- `display.c` and `display.h`
  This component defines function for displaying the word being guessed and the stick figure.

- `hangman.c`
  This component will contain the main function, and any other functions that don't belong in one of the other components (i.e., any functions that don't pertain to reading the wordlist or displaying the word or stick figure).

Global Variables

The wordlist component will contain two global variables related to storing the word list. These may be accessed directly by functions in other components, such as the main() function.

- `words`
  A statically allocated, two-dimensional array of characters for storing the word list. This array will have enough capacity to store 50 words, each up to 20 characters in length.

- `wordCount`
  An integer representing the number of words in the word list. The `words` array has space for 50 words, but we probably won't be using all of them. This variable will keep up with how many words we actually have in the array.

Functions

You'll implement and use the following functions, to break the program into smaller

components and to help simplify main(). You can add more functions if you need them. Functions that are defined in one component and used in another should be prototyped (and documented) in the header.

- `void readWords( char const *filename )`
  This is part of the `wordlist` component. It will read the words in the file and store them in the words array.

- `void displayWord(char word[])`
  This is part of the `display` component. It will print the characters in the "word", some or all of which may be underscores, to standard output, with a single space between each character and no space after the last character.

- `void displayFigure(int numberOfParts)`
  This is part of the `display` component. It will output the stick figure below with the given number of body parts. The head is the first part to be drawn and is represented by an uppercase O. If the number of parts is 2, the head and the torso top should be drawn; if the number of parts is 3, the head, torso top, and torso bottom should be drawn, etc. See the requirements for the order in which the body parts are added to the figure, which corresponds to the number of parts passed to the method.

Command-Line Arguments

For this program, we have to work with command-line arguments. It may be a little while before we get to cover this in class, so I'll go ahead and tell you enough that you should be able to do what's required for this program.

So far, we've written our `main()` routine so that it doesn't take any parameters. Alternatively, you can define `main()` as follows:

```
int main( int argc, char *argv[] )
```

When it's defined this way, the `argc` and `argv` parameters give us access to the arguments passed to the program from the command line. The `argc` (*argument count*) parameter tells how many command-line arguments were given, and the `argv[]` (*argument vector*) parameter is an array of null-terminated strings, one for each of the arguments (we'll learn how to make sense of its type later on). This is a lot like the parameter to `main()` from Java except the number of command-line arguments is given in a separate parameter (`argc`) and the program name itself shows up in the array as argument zero. So, if we run a program as follows:

```
./someProgram this is a test
```

When it starts up, `argc` will have a value of 5. The value of `argv[ 0 ]` will be the string `"./someProgram"`, the value of `argv[ 1 ]` will be `"this"` and so on up to `argv[ 4 ]` with a value of `"test"`.

You will need to use `argc` to see how may command-line arguments are passed to your program, and you'll need to use `argv[ 1 ]` to get the word file name given by the user and `argv[ 2 ]` to get the seed for the random number generator (if there is one).

Random Number Generation

The `hangman` program will "randomly" choose a word for the user to guess. This is done by using the function, `rand`. The index of the word to be guessed may be chosen as follows.

```
int index = rand() % wordCount;
```

Unlike in Java, the rand() function will always provided the *same* sequence of pseudorandom numbers. To make the game more interesting during normal play, you will seed the random number generator with the system time *before using it* - this only needs to be done once. You will need to include the system `time.h` header in order to use the `time` function.

```
srand(time(NULL));
```

To make repeatable testing possible, the `hangman` program takes one optional command-line argument, a positive integer seed. You will need to use the `atoi` (*ascii to integer*) function to convert the command line argument to an integer. If `atoi` returns a positive integer, use it to seed the random number generator via the `srand` function. If it does not return a positive integer, handle the error as described above.

Magic Numbers

Be sure to avoid magic numbers in your source code. Use the preprocessor to give a meaningful name to all the important, non-obvious values you need to use. For example, you can define constants for the maximum length of a word, the maximum number of words, etc.

For constants that are explained right in the line of code where they occur, I wouldn't call these magic numbers. For example, if you need to read two integers, you might write something like the following. Here, the value 2 wouldn't be considered a magic number. It's explained right there in the format string, where we say we'd like to parse two integers.

```
if ( fscanf( stream, "%d%d", &a, &b ) != 2 ) {
    ....
}
```

Build Automation

You get to create your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build your program, compiling each source file to an object file, then linking to produce an executable. Your Makefile should correctly describe the project's dependencies, so if a source or header file changes it rebuilds just the

parts of the project that need to be rebuilt.

Your Makefile should also have a `clean` target that deletes any temporary files made during build or during tests (e.g., executables, objects, program output). A clean rule can look like the following. It doesn't have any prerequisites, so it's only run when it's explicitly specified on the command-line as a target. It doesn't actually build anything; it just runs a sequence of shell commands to clean up the project workspace. Your clean rule will look like the following (we used the <tab> notation to remind you where the hard tabs need to go). In your clean rule, replace things like all-your-object-files with a list of the object files that get built as part of your project.

```
clean:
<tab>rm -f all-your-object-files
<tab>rm -f your-executable-program
<tab>rm -f any-temporary-output-files
<tab>rm -f anything-else-that-doesn't-need-to-go-in-your-repo
```

To use your `clean` target, type the following at the command line, ***but first be sure you are not deleting anything that you need***, as the -f flag forces the removal without asking you if it's OK. You may want to try it first without the -f flag to be sure it is deleting the correct files.

```
$make clean
```

Testing

The starter includes a test script, along with test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other test inputs we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build your program using your Makefile and see how it behaves on the test inputs.

Since this program uses the rand() function, you may need to test it on a common platform system to get the expected output. We've seen rand() give different outputs on different systems, even when seeded with the same value. If your program is working, but it's choosing different words from the word list, be sure you're running it on a common platform system.

You probably won't pass all the tests the first time, and the test script won't work until you have a working Makefile. Until then, you can run tests yourself. You can use a command like the following to compile your programs (although your Makefile will be more efficient, compiling source files separately and then linking them together):

```
$ gcc -g -Wall -std=c99 hangman.c wordlist.c display.c -o hangman
```

I recommend just interacting with your `hangman` program as you try to develop it and debug it. It's easy to do. Just run it from the command prompt using one of the provided wordfiles, animals.txt or food.txt. The example below illustrates "random" behavior in that the random number generator was seeded with the system time in milliseconds instead of a seed provided on the command line. If you ran the game like this, you would most likely get a different word to guess.

```
$ ./hangman food.txt

_ _ _

Remaining letters: a b c d e f g h i j k l m n o p q r s t u v w x y z

letter> a

_ a _

Remaining letters: b c d e f g h i j k l m n o p q r s t u v w x y z

letter> b

 O

_ a _

Remaining letters: c d e f g h i j k l m n o p q r s t u v w x y z

letter> t

 O
 |

_ a _

Remaining letters: c d e f g h i j k l m n o p q r s u v w x y z

letter>
```

If you want to try it out on one of the provided test cases, you can enter commands like the following. Here, we're running the program on one of the test inputs. We're capturing the output to a file and checking the exit status to make sure the program reported a successful execution (most tests should yield successful executions, but some of them test your program's error handling). Finally, we compare the program's output against the expected output.

```
$ ./hangman animals.txt 2 < input-3.txt > output.txt
$ echo $?
0
$ diff expected-3.txt output.txt
```

Here's an example of running the program and capturing output written to stderr:

```
$ ./hangman bad-words-13.txt > output.txt 2> stderr.txt
$ echo $?
1
$ diff stderr-13.txt stderr.txt
```

Sample Input Files

We've prepared several test files to help you make sure your program is working right. These include test inputs read from standard input (input-*.txt) as well as the animals.txt and food.txt files given as command-line arguments with various seed values. There are also several bad word list files (bad-words-*.txt). Here's what these test files do. We've tried to order them so you can use them to help organize your development, working from the earlier, easier test cases to the later ones that test more functionality.

1. This test uses `animals.txt` with a seed of 2 and ends immediately on EOF, so the output is just the initial output and a prompt for a letter.
2. This test uses `animals.txt` with a seed of 2 and enters a single letter found in the word before ending on EOF.
3. This test uses `animals.txt` with a seed of 2 and enters a single letter not found in the word before ending on EOF.
4. This test uses `food.txt` with a seed of 3, enters 7 letters not found in the word, and then the word, no (to not play again).
5. This test uses `food.txt` with a seed of 3, enters all 4 letters in the word, and then the word, no (to not play again).
6. This test uses `animals.txt` with a seed of 2 and enters letters that are in/not in the word, guesses the word, and answers yes (to play again).
7. This test uses `animals.txt` with a seed of 2 and enters letters that are in/not in the word, guesses the word, and answers yellow (to play again).
8. This test uses `animals.txt` with a seed of 2 and enters more than one letter at a time.
9. This test uses `animals.txt` with a seed of 2 and enters a number instead of a letter.
10. This test uses `animals.txt` with a seed of 2 and enters a letter that has been previously guessed.
11. Run with invalid command-line arguments.
12. Run with an wordlist file that doesn't exist.
13. A word list file with too many words.
14. A word list file with a word that is too long.
15. A word list file with a word with non-lowercase letters.

Grading

The grade for your program will depend mostly on how well it functions. We'll also expect it to compile cleanly, to follow the style guide and to follow the design given for the program.

- Compiling cleanly on the common platform: **10 points**
- Working Makefile: **8 points**
- Correct behavior on all tests: **80 points**

- Program follows the style guide: **20 points**
- Deductions
  - Up to **-60 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the p3 directory of your repo. You'll submit by checking files into your repo and pushing the changes back up to the NCSU github.

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't already done this, go back to the assignment for project 2 and follow the instructions for for cloning your repo.

Unpack the starter into your cloned repo

Make sure sure you're in the `p3` directory in your cloned repo. You will need to copy and unpack the project 3 starter. We're providing this as a compressed tar archive, starter3.tgz. After you download this file, you can unpack its contents into your `p3` directory. You can do this like you unpacked the starter from previous projects. Be careful. The starter contains a `.gitignore` file for the new project. Since this file starts with a dot, it will be treated as a hidden file by `ls` and other Unix shell commands. If you unpack elsewhere and then copy into your repo, you may miss this important file. If you `cd` into the `p3` directory of your repo and then unpack, you should get this file and all the other starter files in the right place.

As usual, if you are logged in on one of the common platform systems, you can save yourself a few steps by unpacking the starter directly from our official copy in AFS. Be sure you're in the `p3` directory of your repo and run:

```
$ tar xzvpf /afs/eos.ncsu.edu/courses/csc/csc230/common/www/proj/p3/starter3.tgz
```

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `hangman.c` : main implementation file, created by you.

- `wordlist.c` : wordlist implementation file, created by you.
- `wordlist.h` : wordlist header file, created by you.
- `display.c` : display implementation file, created by you.
- `display.h` : display header file, created by you.
- `Makefile` : the project's Makefile, created by you.
- `input-*.txt`: test inputs, provided with the starter.
- `animals.txt`: animals word list, provided with the starter.
- `food.txt`: food word list, provided with the starter.
- `bad-words-*.txt`: bad word lists, provided with the starter.
- `expected-*.txt`: expected output files, provided with the starter.
- `stderr-*.txt`: expected error output, just for the error test cases, provided with the starter.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file provided with the starter, to tell git not to track temporary files for this project.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Of course, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ unset SSH_ASKPASS # if needed
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to the previous homework. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 3. This job polls

your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O
- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.