Tommy Hollenberg
CS161 Assign6
Design Document
03/6/2017

Understanding the Problem

The purpose of this program is to play Mad Libs, a word game where the player is asked to supply a number of words of given parts of speech without any context. These words are then used to fill in missing words in a predetermined paragraph that match the parts of speech of the words supplied by the player. The resulting paragraph is syntactically correct, but is almost always semantically nonsensical, and can be quite humorous.

This version of the game will allow the player to choose one of three "stories" to fill in, which will be selected by passing a command-line argument of 1, 2, or 3 when running the program. If the user does not pass in any arguments, passes in more than one argument, or passes in an argument that isn't 1, 2, or 3, then the program will notify them of the error in their ways and terminate. The stories that can be chosen from are shown below:

Story 1:
Most doctors agree that bicycle <verb/ing> is a/an <adjective> form of exercise. <verb/ing> a bicycle enables you to develop your <noun> muscles, as well as increase the rate of your <noun> beat. More <nouns> around the world <verb> bicycles than drive <nouns>. No matter what kind of <noun> you <verb>, always be sure to wear a/an <adjective> helmet. Make sure to have <adjective> reflectors too!

Story 2:
Yesterday, <noun> and I went to the park. On our way to the <adjective> park, we saw a <adjective> <noun> on a bike. We also saw big <adjective> balloons tied to a <noun>. Once we got to the <adjective> park, the sky turned <adjective>. It started to <verb> and <verb>. <noun> and I <verb> all the way home. Tomorrow we will try to go to the <adjective> park again and hope it doesn't <verb>.

Story 3:
Spring break 2017, oh how I have been waiting for you! Spring break is when you go to some <adjective> place to spend time with <noun>. Getting to <noun> is going to take <adjective> hours. My favorite part of spring break is <verb/ing> in the <noun>. During spring break, <noun> and I plan to <verb> all the way to <noun>. After spring break, I will be ready to return to <noun> and <verb> hard to finish <noun>. Thanks spring break 2017!

The missing words in each story consist of nouns, verbs, and adjectives. Some of the missing nouns are singular and some are plural. Some of the missing verbs are present participles or gerunds, and therefore require an "-ing" ending.

The user input will be redirected from a "word file" that contains a list of words in the form:

```
<part of speech 1> <word 1>
<part of speech 2> <word 2> etc.
```

There will be no more than 25 words for each part of speech, and each word will be less than 30 characters long. Singular and plural nouns will not be differentiated by the user, but it should be assumed that all nouns ending with 's' are plural and all others are singular. The word used to fill in each blank in the story should be randomly selected from the bank of words of the correct part of speech.

Devising a Plan

The three stories will be hardcoded in a constant static three-dimensional character array. Story 2 has the most segments of text, with 15 (bookending the 14 missing words). The longest segment of text is the first segment of Story 3, which is 101 characters, including the title. Therefore, the story text will be stored in:

```
char story[3][16][102];
```

Each of the three two-dimensional arrays (story[1], story[2], and story[3]) will be terminated by a null C-style string ("\0") for printing purposes.

The parts of speech of the missing words of each story will also be hardcoded in a constant static two-dimensional integer array. Since the largest number of missing words is 14, and each of the three code arrays will be terminated by a -1, the codes will be stored in:

```
int blank_codes[3][15];
```

Part of speech:	Code:
Singular noun	0
Plural noun	1
Non-ing verb	2
-ing verb	3
Adjective	4

The main function will first check that two command-line arguments have been passed in and that the second argument is '1', '2', or '3'. If either of these checks fails, a message instructing the user to fix the problem will be output and the program will end. Otherwise, the story and blank_codes arrays will be defined, the random number generator will be seeded, and word_bank and blanks variables arrays (to hold all the user input words and the words assigned to the story blanks, respectively) will be declared. First, the word_bank will be populated from the word file by calling the **fill_word_bank** function. Next, words of the correct part of speech will be assigned to the story blanks by calling the **assign_words** function. The completed story will be printed to the console by calling the **print_story** function, and finally all memory allocated on the heap will be properly deallocated by calling the **cleanup** function before the program ends by returning 0 to the operating system.

```
int main(number and contents of command-line arguments) {
    If it is not the case that there are two command-line arguments
      and the second is '1', '2', or '3':
        Output message telling user about this requirement.
        Return 0 (end program).
    Seed random number generator with system time.
    Define char story[3][16][102] and int blank_codes[3][15] arrays.
    Declare blanks and word_bank pointers for word arrays.
    Call fill_word_bank, passing the address of the word_bank pointer.
    If the return value of a call to assign_words, passed the correct
      blank_codes and the blanks and word_bank pointers, is false:
        Output "some parts of speech missing".
    Otherwise: call print_story, passing the correct story text and
      the blanks pointer.
    Call cleanup, passing the blanks and word_bank pointers.
    Return 0 (end program)
}
```

The fill_word_bank function will parse input from the word file, allocating memory on the heap for the word_bank and adding words to specific subarrays with the **add_word** function based on their part of speech code as determined by the **get_code** function.

```
void fill_word_bank(address of word_bank pointer in main) {
    Allocate an array of 5 double char pointer (one for each part of
      speech) on the heap and assign the address to word_bank.
    For each element of word_bank:
        Allocate a char pointer on the heap, set it to 0, and assign
          the address to the element of word_bank.
    Declare static arrays of length 10 and 30 to hold the word's part
      of speech and the word itself, respectively.
    Read in a part of speech and word from the file.
    While reading from the file has not failed:
        Declare integer to hold the part of speech code, and assign it
          the return value of a call to get_code, passing the read
          part of speech and word.
        Call add_word, passing the address of the element of word_bank
          with an index matching the part of speech code and the word.
        Read in another part of speech and word from the file.
    }
}
```

```
int get_code(part of speech, word) {
    Declare an integer variable and assign it the length of the word,
      determined by calling strlen.
    If the part of speech is "noun" (using strcmp):
        If the last letter of the word is 's':
            Return 1 (plural noun).
        Return 0 (singular noun).
    If the part of speech is "verb" (using strcmp):
        If the last three letters of the word are "ing":
            Return 3 (-ing verb).
        Return 2 (verb).
    If the part of speech is "adjective" (using strcmp):
        Return 4 (adjective).
    Return -1 (no match).
}

void add_word(word and word_list for the word's part of speech) {
    Declare a "temp" double char pointer and point it to the
      word_list.
    Declare an integer to hold the list_size and initialize it to -1.
    While the (pre-incremented list_size)-th element of word_list
      isn't 0: Do nothing.
    Allocate a char pointer array of length list_size + 2 and assign
      the address to word_list.
    For the first list_size elements of word_list:
        Assign each element of temp to the corresponding element of
          word_list.
    Allocate a char array to hold the word (plus a null terminator)
      and assign the address to the element of word_list with index
      list_size (the (list_size + 1)-th element).
    Copy the word into the char array (using strcpy).
    Set the last element of word_list (the (list_size + 2)-th element)
      at index list_size + 1 to 0.
    Delete[] "temp".
}
```

The assign_words function allocates memory on the heap for the array of missing words and randomly assigns words from the word_bank with the correct part of speech. Since words are assigned to each element of blanks by simply pointing to an element of word_bank, only one call to new[] has been made and therefore only one delete[] is necessary for blanks when deallocating memory in cleanup.

```
bool assign_words(blank_codes, address of blanks pointer in main,
  word_bank) {
    Declare an integer to hold the number of missing words and
      initialize it to -1.
    While the (pre-incremented num_words)-th element of blank_codes
      isn't -1: Do nothing.
    Allocate a char pointer array of length num_words and assign the
      address to blanks.
    For each element of blanks:
        Declare an integer to hold the number of words of the correct
          part of speech in the word_bank and initialize it to -1.
        While the (pre-incremented num_in_bank)-th element of the
          (value of the blank_codes element corresponding to the
          current element of blanks) element of word_bank isn't 0:
            Do nothing.
        If num_in_bank is 0:
            Return false.
        Assign a random element between 0 and num_in_bank - 1 of the
          (value of the blank_codes element corresponding to the
          current element of blanks) element of word_bank to the
          current element of blanks.
    Return true.
}

void print_story(story text, blanks) {
    Declare an integer, i, and initialize it to 0.
    While the (i + 1)-th element of story text isn't a null string:
        Output the i-th element of story followed by the i-th element
          of blanks (the i-th missing word).
        Increment i.
    Output the i-th (final non-null) element of story text.
}

void cleanup(addresses of blanks and word_bank in main) {
    For all 5 part of speech codes:
        For each non-zero C-style string element in the word_bank of
          the part of speech:
            Delete[] the element of the (part of speech)-th element of
              word_bank.
            Set the element of the (part of speech)-th element of
              word_bank to 0.
        Delete[] the part of speech element of word_bank.
        Set the part of speech element of word_bank to 0.
    Delete[] word_bank.
    Set word_bank to 0.
    Delete[] blanks.
    Set blanks to 0.
}
```

Testing Plan

Command-line argument checking:

Input	Result
MadLibs.cpp	"Please pass the desired story number (1,2,3) as the sole command-line argument."
MadLibs.cpp 4	"Please pass the desired story number (1,2,3) as the sole command-line argument."
MadLibs.cpp 1 2	"Please pass the desired story number (1,2,3) as the sole command-line argument."
MadLibs.cpp 3	No output, program proceeds.

get code function:

Input (from word file)	Result
noun chair	0
noun chairs	1
verb run	2
verb running	3
adjective happy	4

Others things to test for:

- The correct story is selected based on the command-line arguments.
- The story is output correctly, with the story text bookending the words filling in the blanks.
- The words filling in the blanks are of the desired parts of speech.
- The words within the correct part of speech are chosen randomly to fill each blank, so the output varies when the program is run multiple times with the same word file.