CS 103 Lab – The Files are *In* the Computer

1 Introduction

In this lab you will modify a word scramble game so that instead of using a "hard-coded" word list, it selects a word from a file. You will learn how to utilize the `valgrind` tool to check for run-time errors in a program and double-check your program's correctness using this tool.

2 What you will learn

After completing this lab you should be able:

- Open and read text files using an ifstream object
- Use fail() to gracefully handle input error conditions
- Dynamically allocate arrays (and deallocate them)
- Review deep vs. shallow copy
- Use `valgrind` to find memory leaks and other memory issues

3 Background Information and Notes

Note: The lecture notes will be a good source of information on how to understand arrays of char pointers and deep vs. shallow copying, which is useful for this lab.

In this lab, we'll give you already-working code for a "word scramble" game (details will be given later). You will change the wordBank array "hard-coded" into the program, and instead allow the user to specify the filename of a text file that contains the words for the word bank. For instance, if `wordbank.txt` is a file containing a list of words, the user would run the program by passing that file name as a command line argument:

```
$ ./scramble wordbank.txt
```

The wordbank file format shall be as follows. The first line shall contain an integer that indicates how many text words follow it. Then exactly that many words should follow separated by some sort of whitespace. See below for an example. (Notice that, in keeping with how >> for cin and ifstreams works, the words can either be on the same line or separate lines as long as there is whitespace between them.

wordbank.txt

```
6
cs103 trojan
midterm
aced
perfect
score
```

4 Procedure

4.1 Starter Files

Start by getting the files for this week:

```
$ cd ~/Dropbox/cs103 # or wherever you like
$ mkdir lab-valgrind
$ cd lab-valgrind
$ wget http://bytes.usc.edu/files/cs103/lab-valgrind.tar
$ tar xvf lab-valgrind.tar
```

Open up `scramble.cpp` in gedit. The given code runs a game that works as follows:

- It defines a "built-in" array `wordBank` of C strings,
- picks a random string from that array
- randomly rearranges its letters
- gives the user 10 guesses to figure out the original, unscrambled string

You should understand the given code before proceeding (though don't worry if you aren't sure of the math underlying the Knuth shuffle function `permute()`, since you won't need to change that.

4.2 Reading Word Bank from a File

Your goal with `scramble.cpp` will be to alter it by deleting the "built-in" words list, instead using any list of words given in a file whose name the user will provide. For example

```
$ ./scramble wordbank.txt # play game using this word list
$ ./scramble palabras.txt # play game in Spanish
```

To get this to work, you should do the following.

1. Include the appropriate file stream header file
2. Delete the global wordBank array declaration and numWords declaration

3. Modify the definition of main() to allow for command line arguments to be accessed **and** add a check to ensure the user has entered enough command line arguments (i.e. provided something that can be used as the filename of their wordbank)
4. Add code to open the file using the command line argument provided as the filename. Check to ensure you could successfully open the file. If you fail to open the file, just output a nice error message and quit the program.
5. Attempt to read an integer from the file which is the number of subsequent words in the file. If you fail to read an integer successfully, close the file and exit gracefully with an appropriate message to the user.
6. Dynamically allocate an array of char*'s named 'wordBank' (one per word that you expect to read from the file) that will point at each character array (i.e. word) that you are about to read in.
7. Declare a character array (a.k.a 'buffer') of 41 characters max to hold the strings that you read in (one by one) from the file [I.e., you may assume the maximum word length will be 40 characters].
8. In a loop, read in each word from the file into the character array you just allocated, then dynamically allocate a new character array that is just the right size to hold that word. Copy the word from the buffer to your new array, and make sure your wordBank has a copy of the pointers to the array you just allocated.
   a. The dynamically allocated memory should be exactly as big as needed. So while the buffer that temporarily holds the word will be 40 bytes large, the dynamically-allocated memory where the words end up should be significantly smaller. An example of this is given in the lecture notes.
9. Close your input file.

The original word scramble code should be able to be reused for everything else (make sure variable names used in your code match those needed for the remainder of the code).

10. At the end of main deallocate all the memory you allocated with new.

Compile your code and test it in English and Spanish as mentioned above.

Next we'll introduce a tool you will use to check your program for bugs.

4.3 Introduction to Valgrind

Just like gdb can be used to help us examine and track down segmentation faults in a program, `valgrind` is a helpful tool to track down memory allocation and usage errors. It will run your program in a special way and tell you if you misuse memory or forget to deallocate it (it also will find other issues, like trying to use garbage values or accessing arrays out of bounds). In this lab we'll be running it like so:

```
$ valgrind --tool=memcheck --leak-check=yes ./scramble wordbank.txt
```

You can see that we are passing a couple of options, then writing the command we want to inspect (including its arguments).

If you used memory appropriately and deallocated everything you should get a summary at the end that looks like this. Note the line that says 10 allocs and 10 frees. This means you dynamically allocated 10 blocks of memory and freed 10. Those numbers should match. If they don't, there was a memory leak.

```
==8146==
==8146== HEAP SUMMARY:
==8146==     in use at exit: 0 bytes in 0 blocks
==8146==   total heap usage: 10 allocs, 10 frees, 8,856 bytes allocated
==8146==
==8146== All heap blocks were freed -- no leaks are possible
==8146==
==8146== For counts of detected and suppressed errors, rerun with: -v
==8146== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

On the other hand, here's an example of what valgrind's output would be if there were an error. Suppose you forget to deallocate the words from the wordbank; here I allocated 10 blocks but only freed 4, which is bad, 6 blocks were not freed! Valgrind uses the the same debug symbols as gdb (the -g option that is baked into `compile`) to even show you which line of code allocated the memory that was never freed (look for the bold type below). It says main() line 48 was where the 'new' operator was used to allocate some memory that was never deleted. This should help you track down what you need to do to delete that memory.

```
==8178==
==8178== HEAP SUMMARY:
==8178==     in use at exit: 40 bytes in 6 blocks
==8178==   total heap usage: 10 allocs, 4 frees, 8,853 bytes allocated
==8178==
==8178== 40 bytes in 6 blocks are definitely lost in loss record 1 of 1
==8178==    at 0x4C2AC27: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8178==    by 0x40109A: main (scramble.cpp:48)
==8178==
==8178== LEAK SUMMARY:
==8178==    definitely lost: 40 bytes in 6 blocks
==8178==    indirectly lost: 0 bytes in 0 blocks
==8178==      possibly lost: 0 bytes in 0 blocks
==8178==    still reachable: 0 bytes in 0 blocks
==8178==         suppressed: 0 bytes in 0 blocks
==8178==
==8178== For counts of detected and suppressed errors, rerun with: -v
==8178== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

4.4 Fixing Valgrind Errors

A program `memleak.cpp` is included with the files this week. It basically contains two different tests, both of which have bugs.

Build and run it (your input in ***bold italics***):

```
$ ./compile memleak
$ ./memleak 1
Test 1 sum is 475
$ ./memleak 2
Enter a word: rutabaga
...
```

Both of these tests are wrong, but you will now fix them.

IMPORTANT: For all of the bugs you fix in this part of the lab, you have to keep some documentation of what bugs you find and what you change to fix them. This is mandatory because you will review with your CP/TA when you "check in." We recommend that you:

- **prominently write comments in your code indicating each line you change and why you changed it**
- **keep a separate list somewhere of the chronological order of changes (was it for memleak 1 or memleak 2?) to guide your discussion with the CP/TA**

1. Run the 1st test:

   ```
   $ ./memleak 1
   ```

   This program is supposed to be adding up 7 random numbers between 0 and 99 and printing out the sum. However, now let's see if there are any hidden memory issues. Let's run valgrind:

   ```
   $ valgrind --tool=memcheck --leak-check=yes ./memleak 1
   ```

   You should see 2 errors (an Invalid Read and a block of memory that is definitely lost). An invalid read means you are trying to access a piece of memory that is not allocated to you. Usually it's because you have a bad pointer or access things beyond the end of an array.

   Use the line numbers in the error messages as clues and try to fix the code (Google the error message if you are having trouble understanding it). Recompile `memleak.cpp` and rerun `valgrind` until you have 0 errors.

2. Run the 2nd test:

   ```
   $ ./memleak 2
   ```

   This program should ask you for a word, then print it, and print it in reverse. You may see a spew of text as the program completes. Scroll to the top and you'll see a message like:

   ```
   *** glibc detected *** ./memleak double free or corruption…
   ```

   This is indicative of dynamic memory problems. Let's run valgrind

   ```
   $ valgrind --tool=memcheck --leak-check=yes ./memleak 2
   ```

   You should see 3 errors. Try to read the messages (and line numbers) carefully to understand what it is saying. Some errors may be due to the same root cause. So as you fix one, re-run to see if it helps.

   Recompile and rerun valgrind until you have 0 errors.

4.5 Rerun Word Scramble through Valgrind

Re-run your `scramble.cpp` program (word scramble with word bank code) through valgrind and ensure there are no errors.

```
$ valgrind --tool=memcheck --leak-check=yes ./scramble wordbank.txt
```

Demonstrate your working code to your TA/Sherpa and change a few words in the wordbank to demonstrate the working file I/O. Also, make sure it has no memory errors or leaks by using `valgrind`.

You should use valgrind in every lab and homework assignment from now on because it will help you check for:

- Memory leaks (forgetting to deallocate) and corruption
- Using arrays out-of-bounds
- Accidentally using garbage values