CS246—Assignment 2 (Spring 2017)

Due Date 1: Friday, May 26, 5pm
Due Date 2: Friday, June 2, 5pm

Questions 1, 2a, 3a, 4a, 5a are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

Note: On this and subsequent assignments, you will be required to take responsibility for your own testing. As part of that requirement, this assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. If you look at the deliverables and their due dates, you will notice that there is *no* C++ code due on Due Date 1. Instead, you will be asked to submit test suites for C++ programs that you will later submit by Due Date 2.

Test suites will be in a format compatible with A1Q4/5/6. So if you did a good job writing your `runSuite` script, it will serve you well on this assignment.

Be sure to do a good job on your test suites, as they will be your primary tool for verifying the correctness of your submission.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.

Note: Further to the previous note, your solutions may only `#include` the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, and `<string>`. No other standard headers are allowed. Marmoset will check for this.

Note: There will be a handmarking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. Please code to a standard that you would expect from someone else if you had to maintain their code. Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/current/AssignmentGuidelines.shtml`

Note: You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do. A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

1. **Note: there is no coding associated with this problem**. You are given a non-empty array $a[0..n-1]$, containing n integers. The program `SORT` sorts the array in ascending order. The output of the program is a print of the sorted array starting from the smallest element, with one element printed per line. For example, if the input is

   ```
   -9 4 5 -1 3 10
   ```

   then `SORT` prints

   ```
   -9
   -1
   3
   4
   5
   10
   ```

   Your task is not to write this program, but to design a test suite for this program. Your test suite must be such that a correct implementation of this program passes all of your tests, but a buggy implementation will fail at least one of your tests. Marmoset will use a correct implementation and several buggy implementations to evaluate your test suite in the manner just described.

   Your test suite should take the form described in A1P5: each test should provide its input in the file `testname.in`, and its expected output in the file `testname.out`. The collection of all `testname`s should be contained in the file `suiteq1.txt`.

   (a) **Due on Due Date 1**: Zip up all of the files that make up your test suite into the file `a2q1.zip`, and submit to Marmoset.

2. In this question you will encrypt and decrypt text using a shift cypher. A shift cypher is an ancient encryption technique in which each letter in the text to be encrypted is replaced by a letter some fixed number of positions down/up the alphabet. While the input to a shift cypher can be any sequence of characters, we will restrict ourselves to a shift cypher that only encrypts upper or lower case english alphabetic characters. All other characters will be unchanged in the output. The decryption process is the reverse of the encryption process.

   The program you are to write should be runnable with 0, 1 or 2 arguments. If no argument is provided, then the cypher shifts 3 to the right e.g. A becomes D, Z becomes C etc. If only 1 argument is provided, then it must be an integer value between 0 and 25 inclusive and represents the shift value. If a second argument is also provided, it is either the string "left" or the string "right" indicating the direction in which the shift cypher operates. It is not possible to provide a shift direction without a shift value. If the user provides arguments that do not satisfy these conditions, the program must print the string "ERROR" to standard error and exit with a non-zero status code (an additional descriptive error message can be displayed but is not needed). In the case of an error, no output is generated on standard output.

   Implementation help: In the a2 directory you will find a program called `args.cc`, which demonstrates how to access command line arguments from a C++ program. Use that program as an example to help you solve this problem.

   The input to the program is from standard input. The input is formatted as follows: each line begins with either the character 'e' or 'd' or 'q'. You may assume that no input line will begin with any other character (i.e. you need not check for invalid input in this case). If the input is 'e', all characters following the 'e' are processed for encryption until a newline is encountered. If the input is 'd' the program processes all characters following the 'd' for decryption until

a newline is encountered. The encrypted/decrypted output is sent to standard output on its own line. 'q' quits the program. Following is a sample interaction with the program when executed without any arguments (output in italics):

```
eHelloWorld!
```
KhoorZruog!
```
dKhoorZruog!
```
HelloWorld!
```
q
```

Implementation help: A compiled binary of a correctly implemented solution is provided. You can use it to resolve any ambiguities in the problem requirements as well as generating your test suite. Two sample test cases are also provided.

(a) **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq2.txt`. Zip your suite file, together with the associated `.in`, `.out` and `.args` files, into the file `a2q2.zip`.

(b) **Due on Due Date 2**: Write the program in C++. Save your solution in `a2q2.cc`.

3. A *prettyprinter* is a tool that takes program source code as input and outputs the same code, nicely formatted for readability. In this problem, you will write a prettyprinter for a C-like language.

   The input for your program will be a sequence of "words" on stdin, spanning one or more lines. The words denote *tokens*, the "pieces" that make up a program. The words will be separated from each other by one or more whitespace characters (space, tab, newline). Your program will take these tokens and arrange them nicely on stdout, according to the following rules:

   - Initially, the code is flush to the left margin (i.e., not indented);
   - If the word is `;`, print the word and go to the next line;
   - If the word is `{`, print the word, go to the next line, and the following lines will be indented by one more space than previously;
   - If the word is `}`, it should be printed on its own line, indented one character to the *left* of the lines between it and its matching `{` (i.e., the indentation level will be the same as the indentation level of the line that contained the matching `{`), and the following lines are indented to the same level as this word;
   - If the word is `//`, then the rest of the current line of input is considered a comment, and must be printed *exactly* as it is, including spacing;
   - Except for comments, all of the tokens on a line should be separated from one another by exactly one space.

   Sample input:

```
int f ( int x ) { // This is my function
int y = x ; y = y + 1 ; return y ; }     // This is the   END   of my function
int main () { int n = 0 ; while ( n < 10 ) { n = f ( n ) ; } }
```

   Corresponding output:

```
int f ( int x ) {
 // This is my function
 int y = x ;
 y = y + 1 ;
```

```
 return y ;
}
// This is the   END   of my function
int main () {
 int n = 0 ;
 while ( n < 10 ) {
  n = f ( n ) ;
 }
}
```

Your solution must not print any extra whitespace at the end of the line (exception: if a comment ends with spaces, then you must keep those spaces in your output). However, if trailing space is the only thing wrong with your program, you can receive partial credit.

You may assume: That all tokens are separated by whitespace. In particular, the special words `{`, `}`, `;`, and `//` will not be "attached" to other tokens, as they can be in C. You may also assume that each right brace `}` has a "matching" left brace `{`.

You may not assume: That the input language is actually C. All you are told is that the input language uses brace brackets, semicolons, and `//` comments in a way similar to C, but subject to those constraints, the input could be anything. So do not assume that any properties of the C language, beyond what you have been told, will be true for the input.

Implementation help: A compiled binary of a correctly implemented solution is provided. You can use it to resolve any ambiguities in the problem requirements as well as generating your test suite. A sample test case is also provided.

(a) **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq3.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q3.zip`.

(b) **Due on Due Date 2**: Write the program in C++. Save your solution in `a2q3.cc`.

4. In this problem, you will implement your own version of the Linux program wc which is short for word count.

   Your wc program, when given no command line arguments, should evaluate input received from standard input and print the number of lines, words, and characters read until an end-of-file signal is received. This default behaviour can be modified by giving wc one or more of the following command-line arguments:

   - `-l`: print the number of lines present
   - `-w`: print the number of words present
   - `-c`: print the number of characters present
   - `file-name`: print the evaluation of the contents of `file-name`

   Note that command line arguments such as `-l`, `-w` and `-c` are often referred to as flags or options as they are used to change the behaviour of a program. If any of `-l`, `-w`, or `-c` are present, then the values for the flags which are not present are not printed. For example, `wc -c -l` prints the number of lines followed by the number of characters which are read and NOT the number of words. If none of these flags are present, the program reverts to the default of printing lines, words, and characters.

When no file-names are provided, the program generates one line of output:

```
num-lines num-words num-chars
```

Note that irrespective of the order in which the flags are specified on the command line, the output is always generated in the order mentioned above i.e. first lines, then words and then characters with any of these removed as discussed earlier. The first number printed is always left justified and a single space separates each value that is printed afterwards. There is no space after that last value. (**Note that this is different from how the Linux wc program formats its output**).

In the case that one or more file-names are specified on the command line, input is not read from standard input. Instead, the program evaluates the contents of the files, in the order they occur as arguments to the program. For each file that is evaluated, the program generates one line of output with the format:

```
num-lines num-words num-chars file-name
```

If more than one files were given as command line arguments, after the output for each of these files has been printed, one additional line is printed that displays the total number of lines, words, and characters followed by the string "total". If a file-name is provided twice, the file is evaluated twice. You may assume that all files passed as arguments exist and are readable.

One interesting feature is that command-line arguments can occur in any order and can each occur multiple times. For instance, the following is a valid call to the program:

```
wc -w file1.txt -l file2.txt file1.txt -l
```

Implementation help 1: To get you started, we have provided you with two structure definitions and some functions which will make tracking the information easier. The struct `Options` is designed to keep track of which types of outputs are expected and the names of each file being read. The struct `Counts` is designed to keep track of the number of lines, words, and characters present in an input and the name of the input. These structures and functions can be found in **a2/a2q4.cc**. You do not have to use either and you can change them as you need to complete this question.

Implementation help 2: A compiled binary of a correctly implemented solution is provided. You can use it to resolve any ambiguities in the problem requirements as well as generating your test suite.

(a) **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq4.txt`. Zip your suite file, together with the associated `.in`, `.out`, `.args` and any other text files, into the file `a2q4.zip`.

(b) **Due on Due Date 2**: Write the program in C++. Save your solution in `a2q4.cc`.

5. We typically use arrays to store collections of items (say, integers). We can allow for limited growth of a collection by allocating more space than typically needed, and then keeping track of how much space was actually used. We can allow for unlimited growth of the array by allocating the array on the heap and resizing as necessary. The following structure encapsulates a partially-filled array:

```
struct IntArray {
  int size;  // number of elements the array currently holds
  int capacity;  // number of elements the array could hold, given current
                 // memory allocation to contents
  int *contents;
};
```

- Write the function `readIntArray` which returns an `IntArray` structure, and whose signature is as follows:

  ```
  IntArray readIntArray();
  ```

  The function `readIntArray` consumes as many integers from `cin` as are available, populates an `IntArray` structure in order with these, and then returns the structure. If a token that cannot be parsed as an integer is encountered before the structure is full, `readIntArray` stops attempting to read more integers and only fills as much of the array as needed, leaving the rest unfilled. If a non-integer is encountered, the first offending character should be removed from the input stream (i.e., call `cin.ignore` once with no arguments). In all circumstances, the field `size` should accurately represent the number of elements actually stored in the array and `capacity` should represent the amount of storage currently allocated to the array.

- Write the function `addToIntArray`, which takes a pointer to an `IntArray` structure and adds as many integers to the structure as are available on `cin`. The behaviour is identical to `readIntArray`, except that integers are being added to the end of an existing `IntArray`. The signature is as follows:

  ```
  void addToIntArray(IntArray&);
  ```

- Write the function `printIntArray`, which takes a pointer to an `IntArray` structure, and whose signature is as follows:

  ```
  void printIntArray(const IntArray&);
  ```

  The function `printIntArray(a)` prints the contents of `a` (as many elements as are actually present) to `cout`, on the same line, separated by spaces, and followed by a newline. There should be a space after each element in the array (including the last element), and not before the first element.

  It is not valid to print or add to an array that has not previously been read, because its fields may not be properly set. You should not test this.

For memory allocation, you **must** follow this allocation scheme: every `IntArray` structure begins with a capacity of 0. The first time data is stored in an `IntArray` structure, it is given a capacity of 5 and space allocated accordingly. If at any point, this capacity proves to be not enough, you must double the capacity (so capacities will go from 5 to 10 to 20 to 40 ...). Note that there is no `realloc` in C++, so doubling the size of an array necessitates allocating a new array and copying items over. Your program must not leak memory.

Implementation help 1: A test harness is available in the starter file `a2q5.cc`, which you will find in your `a2` directory. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use).

Implementation help 2: A compiled binary of a correctly implemented solution is provided. You can use it to resolve any ambiguities in the problem requirements as well as generating your test suite. One sample test case is also provided.

(a) **Due on Due Date 1:** Design a test suite for this program, using the main function provided in the test harness. Call your suite file `suiteq5.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q5.zip`.

(b) **Due on Due Date 2:** Write the program in C++. Call your solution `a2q5.cc`.