Assignment #2

Due Date 1: Friday, 28 September, 2018, 5:00 pm
Due Date 2: Friday, 5 October, 2018, 5:00 pm

- **Questions 1, 2a, 3a, 4a, 5a are due on Due Date 1; the remaining questions are due on Due Date 2.**
- On this and subsequent assignments, you will take responsibility for your own testing. 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 that of the latter questions of Assignment 1, so if you did a good job writing your `runSuite` script, that experience will serve you well here.
- Design your test suites with care; they are your primary tool for verifying the correctness of your code. Note that test suite submission zip files are restricted to contain a maximum of 40 tests, and the size of each file is also restricted to 300 bytes; this is to encourage you not combine all of your testing eggs in one basket.
- You must use the standard C++ I/O streaming and memory management (MM) facilities on this assignment; you may **not** use C-style I/O or MM. More concretely, you may `#include` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `iomanip`, and `string`. Marmoset will be setup to **reject** submissions that use C-style I/O or MM, or libraries other than the ones specified above.
- We will manually check that you follow a reasonable standard of documentation and style, and to verify any assignment requirements that are not automatically enforced by Marmoset. 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`
- We have provided some code and executables in the subdirectory `codeForStudents`.
- **You may not ask 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.

Part 1

Note: You do not have to write any C++ code for this problem.

A gas-heated house uses more gas in the winter than in the summer; to keep the monthly gas expenses roughly the same from month to month, the gas company offers an "equal billing" plan. Under such a plan, the customer pays the same amount every month, and then once every six months (i.e., months 6, 12, 18, etc.) is "true-up month". In true-up month, if the customer owes more than the monthly amount, the customer must pay the entire outstanding balance, and the billing cycle starts over. On the other hand, if the customer has used less gas than she was charged for, the gas company does not refund the money; instead, it charges 0 for as many months as it takes to square up the account. Specifically, if the customer's overpaid balance is greater than the monthly rate, the gas company charges 0 for that month, and reduces the overpaid balance by one monthly amount. If the customer's overpaid balance is positive, but less than the monthly rate, the gas company charges the difference between the monthly rate and the balance. Once the overpaid balance has been reduced to 0, normal monthly billing resumes.

The other thing that happens in true-up month is a new monthly rate is calculated for the next billing cycle. This will be the average monthly use since the last true-up month, rounded down, plus one.

The program in this problem takes the initial monthly billing amount as a command line argument. The input (on stdin) is the sequence of monthly amounts used (starting with month 1). The output is the amount of the monthly bill for each month, with some explanatory information, as shown below. EOF ends the program. An example run of the program appears below (input is on the left; corresponding output is on the right):

```
./gas 10
8                       You pay: 10  (Usage: 8  Paid: 10  Balance: 0)
8                       You pay: 10  (Usage: 16  Paid: 20  Balance: 0)
8                       You pay: 10  (Usage: 24  Paid: 30  Balance: 0)
8                       You pay: 10  (Usage: 32  Paid: 40  Balance: 0)
8                       You pay: 10  (Usage: 40  Paid: 50  Balance: 0)
8                       You pay: 0  (Usage: 48  Paid: 0  Balance: 2)
                        Rate now 9
8                       You pay: 7  (Usage: 8  Paid: 7  Balance: 0)
8                       You pay: 9  (Usage: 16  Paid: 16  Balance: 0)
8                       You pay: 9  (Usage: 24  Paid: 25  Balance: 0)
8                       You pay: 9  (Usage: 32  Paid: 34  Balance: 0)
```

Your task is *not* to write such a program, but to design a test suite for one, stored as a plain text file named `suiteq1.txt`. Your test suite must be such that a correct implementation of this program will pass 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.

Due on Due Date 1: Submit a file called `a2q1.zip` that contains the test suite you designed, called `suiteq1.txt`, and all of the `.in`, `.out`, and `.args` files.

Part 2

In this question, you will encrypt and decrypt text using a *shift cypher*, 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 en/de-crypting only upper or lower case English alphabetic characters; that is, any other characters that appear in the input will be left 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` (and `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.

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. You may use any part of that code in solving this problem.

The input to the program is from standard input. The input is formatted as follows: each line begins with the character 'e', 'd', or 'q'. You may assume that no input line will begin with any other character. 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. If an EOF is encountered, the program quits. 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:** Submit a file called `a2q2.zip` that contains the test suite you designed, called `suiteq2.txt`, and all of the `.in`, `.out`, and `.args` files.

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

Part 3

In this problem, you will write a program called `wordWrap`, whose purpose is to confine text to a given width with the desired justification. More specifically, `wordWrap` takes a sequence of tokens on stdin and echoes them to stdout, such that the width of the output is no wider than the provided command-line argument and the output is left justified, centered, or right justified. The desired width is provided to `wordWrap` by an optional `-w textWidth` argument in which `textWidth` is a positive integer denoting the width of the text. If no width is supplied on the command line, the default text width is 25. Furthermore, `wordWrap` accepts a second optional argument indicting the desired justification. This argument could be `-l`, `-c` or `-r` which correspond to *left justified*, *centered* and *right justified* respectively. If no justification is provided, the output text is left justified.

For example, suppose the command `wordWrap -w 20 -l` is executed and the text is as seen below:

```
Friends Romans countrymen lend me your ears I come to bury Caesar not
to praise him
```

then the output would be:

```
Friends Romans
countrymen lend me
your ears I come to
bury Caesar not to
praise him
```

On the other hand, the command `wordWrap -w 20 -r` with the same input as before would generate the output:

```
      Friends Romans
  countrymen lend me
 your ears I come to
  bury Caesar not to
          praise him
```

If a word is too long to fit on what remains of the line, put it on the next line. Do not break a word unless it is longer than the entire allowed width. For example, the same text with a width of 8 becomes

```
Friends
Romans
countrym
en lend
me your
ears I
come to
bury
Caesar
not to
praise
him
```

When outputting words, separate them by a single whitespace character, either a single space or a single newline, regardless of how they are spaced in the input. For example, if the input contained words separated by two spaces, they would still be separated by one whitespace character in the output. Don't worry about odd cases that make no sense, like someone specifying both `-l` and `-r` on the command line.

a) **Due on Due Date 1:** Submit a file called `a2q3.zip` that contains the test suite you designed, called `suiteq3.txt`, and all of the `.in`, `.out`, and `.args` files.

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

Part 4

Implement a simplified version of the Unix command `wc`; have a look at the man page, and try out the real version. Your implementation should be able to take input from either a single file name specified on the command line, or from stdin. You are to support the flags `-c`, `-l`, and `-w`. You can assume each flag appears at most once, and you don't have to worry about supporting combinations like `-wc` or `-lcw` as one token. You may also assume (without checking) that all input is valid, and that any file named on the command line exists and is readable. Your output may differ from that of the "real" `wc` with respect to whitespace usage, but the counts you print should match. If none of the `-l`, `-w`, or `-c` flags are specified, then print all three counts, just like the real `wc` does. The order of the output should be: *[num lines] [num words] [num chars] [filename]*, although not all of the numbers may be asked for on a given run.

Here are some example test runs created using the first output file from the previous question (`"Friends, ..."`):

```
$ mv a.out my_wc
$ ./my_wc text1.in
5  16  84  text1.in
$ ./my_wc -l -w -c text1.in
5  16  84  text1.in
$ ./my_wc -l -w -c < text1.in
5  16  84
$ ./my_wc -c -w  text1.in
16  84  text1.in
$ ./my_wc -c -l text1.in
5  84  text1.in
```

a) **Due on Due Date 1:** Submit a file called `a2q4.zip` that contains the test suite you designed, called `suiteq4.txt`, all of the `.in`, `.out`, `.args` files, and any other files you used in your testing.

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

Part 5

We typically use arrays to store collections of items (say, booleans). 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 simulates binary numbers by encapsulating a partially-filled array:

```
struct BinaryNum {
    int size;         // number of elements the array currently holds
    int capacity;     // number of elements the array could hold,
                      // given current memory allocation to contents
    bool *contents;   // (pointer to) heap-allocated array of bools
};
```

- Write the function `readBinaryNum` which returns a `BinaryNum` structure, and whose signature is as follows:

  ```
  BinaryNum readBinaryNum();
  ```

  The function `readBinaryNum` consumes as many ones and zeroes from `cin` as are available, populates a `BinaryNum` structure in order with these, and then returns the structure. If a non-whitespace character that is not a one or a zero is encountered before the structure is full, then `readBinaryNum` fills as much of the array as needed, leaving the rest unfilled. If a non one or zero character is encountered, the first offending character should be removed from the input stream. 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.

  HINT: Store bit 0 (the first one you read in) in array element 0, etc.

- Write the function `binaryConcat`, which takes a `BinaryNum` structure by reference whose signature is as follows:

  ```
  void binaryConcat(BinaryNum &);
  ```

  The function `binaryConcat` concatenates on to the end of the structure passed in as many booleans as are available on `cin`. The behaviour is identical to `readBinaryNum`, except that integers are being added to the end of an existing `BinaryNum`.

- Write the function `binaryToDecimal`, which takes a `BinaryNum` structure by reference to const and returns an int. The signature is as follows:

  ```
  int binaryToDecimal(const BinaryNum &);
  ```

  The function `binaryToDecimal` should return the decimal value equivalent of the binary number represented by the given BinaryNum structure. The behaviour of this function is undefined if the decimal equivalent of the binary number exceeds the maximum value that can be stored within an int.

- Write the function `printBinaryNum`, which takes a `BinaryNum` structure by reference to const and a separator character, and whose signature is as follows:

  ```
  void printBinaryNum(std::ostream &out, const BinaryNum &, char);
  ```

  The function `printBinaryNum(out, binNum, sep)` prints the contents of `binNum` (as many elements as are actually present) to the given stream `out`, on the same line, separated by characters denoted by `sep`, and followed by a newline. There should be a separator character in-between each element in the array, this means there should be none before the first element or after the last element.

  It is not valid to perform any operations on a BinaryNum that has not first been read, because its fields may not be properly set. You should not test this.

- Write the function for the left-shift operator `<<` which takes a left hand operand `BinaryNum` structure by reference and a right hand operand int. The function returns a reference to the same `BinaryNum` structure that is passed as an argument. The signature is as follows:

  ```
  BinaryNum &operator<<(BinaryNum &, int);
  ```

  Assuming `binNum` is a BinaryNum variable and `x` is an int, the expression `binNum << x` (or equivalently `operator<<(binNum,x)`) will move each bit in `binNum` exactly `x` places to the left adding the appropriate number of zeroes onto the end of the number. As an example a BinaryNumber that when printed shows `1011011` if shifted left by 3 will then show `1011011000`.

For memory allocation, you **must** follow this allocation scheme: every `BinaryNum` structure begins with a capacity of 0. The first time data is stored in a `BinaryNum` structure, it is given a capacity of 4 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 4 to 8 to 16 to 32 ...). Note that there is no `realloc` in C++, so doubling the size of an array necessitates allocating a new array and copying items over. **If you do not follow this allocation scheme, you will not receive any correctness marks.** Also, you will lose marks if your solution leaks memory.

A header file and test harness are available in the starter files `binarynum.h` and `a2q5.cc`, which you will find in your `cs246/1189/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.** A sample test case that can be run using the test harness is also provided. 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).

a) **Due on Due Date 1:** Submit a file called `a2q5.zip` that contains the test suite you designed, called `suiteq5.txt`, and all of the `.in`, `.out`, and `.args` files. Note that `suiteq5.txt` should use the `main` function provided in the test harness we gave you.

b) **Due on Due Date 2:** Implement the required functions in the file `binarynum.cc`. Submit this file, the provided header, the provided test harness, and a `Makefile` in the file `a2q5.zip`. Your `Makefile` must build an executable called `a2q5`.