Assignment #2

Due Date 1: Friday, 1 February, 2019, 5:00 pm
Due Date 2: Friday, 8 February, 2019, 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 credit card allows the owner to complete purchases, which add to the outstanding balance. Each month, the balance should be payed off. If it's not payed off in full, then an interest is applied and added to the balance. Thus, at the end of each month, the new balance is calculated as:

$New\ balance = Previous\ balance + Interest + Purchases - Payment$

The program in this problem takes the initial balance and the annual interest rate (percentage) as command line arguments. The monthly interest rate should be calculated by the program and is 1/12 of the annual rate. The calculated interest each month is just the previous balance multiplied by the interest rate. The total amount of purchases and the payment each month are provided by the user as the input (on stdin).

After the user input for each month, the program must print all the values (previous balance, interest, purchases, payment, and new balance) to the output, as shown below. All numbers are to be printed with two fixed decimal places. An EOF in the input ends the program.

An example run of the program appears below. The numbers marked in blue (500, 800, 755.25, and 350.5) are user input, the remaining text and numbers are the program output:

```
./creditCardCalculator 10000 18
Creadit Card Calculator
Initial Balance: $10000.00
Interest: 18.00%/year (1.50%/month)

Enter the amount of the new purchases this month: 500
Enter the amount paid this month: 800

Month: 1
Previous balance: $10000.00
 + Interest: $150.00 + Purchases: $500.00 - Payment: $800.00
 = New balance: $9850.00

Enter the amount of the new purchases this month: 755.25
Enter the amount paid this month: 350.5

Month: 2
Previous balance: $9850.00
 + Interest: $147.75 + Purchases: $755.25 - Payment: $350.50
 = New balance: $10402.50

Enter the amount of the new purchases this month: <EOF>
```

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.

We have provided a sample solution to this problem, in the form of an executable binary, in your a2 directory. Note that it is compiled to run on the student.cs environment. We will not provide other versions (e.g. for Windows) of this executable. You can use this binary, together with your produceOutputs script from A1, to generate the outputs for your chosen inputs.

Your test suite should take the form described in A1Q6: each test should provide its input and arguments in the files `testname.args` and `testname.in`, and its expected output in the file `testname.out`. The collection of all testnames should be contained in the file `suiteq1.txt`.

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 we will implement an election that uses *cumulative voting*. In this voting scheme, a voter has X number of votes that they can choose to distribute among the candidates which are numbered from 1 to n, where n is at most 10 (the edge case of 0 candidates is possible and is considered valid input). The value for X may be provided as an optional positive command line argument. If a value of X is not provided as an argument, the default value of n is used. Input begins with the names of candidates (one full name per line). The first name is considered as candidate 1, and second as candidate 2, and so on. A candidate's name will never contain a numeral and consists of at least 1 and at most 15 characters (including any spaces). The list of candidates is followed by some number of lines where each line indicates one voter's distribution of votes, which we call a ballot. For a ballot the i^{th} column indicates the number of votes allocated to the i^{th} candidate. A ballot is considered invalid (*spoilt*) if it does not consist of n columns or the sum of the votes in the ballot exceeds X. In addition, the votes allocated to a specific candidate within a ballot are always non-negative. The number of voters is unknown beforehand, but is, of course, non-negative. Votes are terminated by end-of-line.

As an example, given the following data:

```
Victor Taylor
Denise Duncan
Kamal Ramdhan
Michael Ali
Anisa Sawh
Carol Khan
Gary Owen
3 0 1 0 0 1 2
1 1 1 1 0 1 2
1 1 1 1 1 1 1
2 1 3 1
7 0 0 0 0 0 0
1 1 1 1 1 1 2
```

your program should produce the following output:

```
Number of voters: 6
Number of valid ballots: 4
Number of spoilt ballots: 2

Candidate      Score

Victor Taylor   12
Denise Duncan    2
Kamal Ramdhan    3
Michael Ali      2
Anisa Sawh       1
Carol Khan       3
Gary Owen        5
```

Use the following format for the data in the two columns:

- Candidate: left-justified, 15 characters wide
- Score: right-justified, 3 characters wide

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.

Implementation help: A compiled binary of a correctly implemented solution is provided (`a2q2`). 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 (`q2.in` and `q2.out`).

Note: Do not copy paste the example above to create a test file. The formatting might NOT be correct. Use the test case provided to you (`q2.in` and `q2.out`) within the a2 directory.

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

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.

a) **Due on Due Date 1:** Design a test suite for this program. 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 program similar to the Linux command `head` (have a look at the man page, and try out the real version). Your implementation should be able to take input from either one or more files specified on the command line, or from `stdin`. If no command line arguments are provided, the program should print the first 10 lines of each file to standard output. The program must also support two command line arguments: `-n` and `-s`:

```
-n [NUM]
   If this argument is specified, then the program should print the first NUM lines
   of each file instead of the first 10

-s [NUM]
   If this argument is specified, then the program should skip the first NUM lines
   of each file, before start printing the number of lines specified by -n
   (or the default 10 lines).
```

For example:

```
./head -n 20 -s 5 myfile.txt
   should skip the first 5 lines of myfile.txt, then print the next 20 lines to
   the standard output

./head
   should read from stdin and write the first 10 lines read to stdout
```

a) **Due on Due Date 1:** Design a test suite for this program. 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. 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. Sometimes, we'd like to keep track of our free memory. The following two structures are used to keep track of free "chunks" of memory stored in "pools".

```
struct Chunk {
    size_t length; // Length of this chunk.
    int *mem; // Array of length integers.
};

struct Pool {
    size_t capacity; // Capacity is the size of the chunks array.
    size_t size; // length is the number of elements in the chunks array.
    Chunk *chunks; // Array of chunks.
};
```

- Write the function `readPool` which returns a `Pool` structure by value, and whose signature is as follows:

  ```
  Pool readPool();
  ```

  The function `readPool` consumes as many integers (you may assume are greater than zero) from `cin` as are available which represent the sizes of chunks populating a `Pool` structure in order with chunks of these size, and then returns the structure. When a non-whitespace character that is not an integer is encountered before the structure is full, then `readPool` fills as much of the array as needed, leaving the rest unfilled. When an offending 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 chunks actually stored in the array and `capacity` should represent the amount of storage currently allocated to the array.

  NOTE: Your chunks must have valid pointers to heap-allocated arrays of integers of the appropriate size!

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

- Write the function `addChunk`, which takes a `Pool` structure by reference, and a `size_t` representing the length of the chunk to be added whose signature is:

  ```
  void addChunk(Pool &, size_t);
  ```

  The function `addChunk` adds a chunk to the end of the pool of the corresponding length.

- Write the function `printPool`, which takes an ostream reference, and a `Pool` structure by reference to const, and whose signature is as follows:

  ```
  void PrintPool(std::ostream &out, const Pool &);
  ```

  The function `printBool(out, pool)` prints the lengths of the individual chunks stored in the `Pool` to `out`, on the same line, with a space in between each length. This means there should be none before the first chunk-length or after the last chunk-length.

- Write the function `findFreeMem`, which takes a `Pool` structure by reference, and a `size_t` representing the desired minimum space, and produces the first chunk in the Pool with enough space (equal to or greater than the provided `size_t`).

  If such a `Chunk` is found, it should be returned by value, and should be removed from the array of chunks in the `Pool`. This means the size of the `Pool` is decreased by one, and every `Chunk` in the array after the removed `Chunk` is moved left in the array one space.

  If no such `Chunk` exists (there is not one with enough space), then you should return by value a `Chunk` with length 0.

For memory allocation of `Pool`'s array of chunks you **must** follow this allocation scheme: every `Pool` structure begins with a capacity of 0. The first time data is stored in a `Pool` 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 `pool.h` and `pool_test_harness.cc`, which you will find in your `cs246/1191/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 `pool.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`.