CS246—Assignment 2 (Fall 2016)

J. Avery, C. Kierstead and B. Lushman

Due Date 1: Friday, September 30, 5pm
Due Date 2: Friday, October 7, 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 a few short answer questions and test suites for C++ programs that you will later submit by Due Date 2.

Test suites will be in a format compatible with A1Q5/6. 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. Instead, we will provide compiled executables, suitable for running on `linux.student.cs`, that you can use to check intended behaviour. 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 `maxSum` determines the indices i and j, $i \leq j$, for which $\sum_{k=i}^{j} a[k]$ is maximized and reports the maximum value of $\sum_{k=i}^{j} a[k]$. Note that since $i \leq j$, the sum always contains at least one array element. For example, if the input is

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

   then `maxSum` prints

   ```
   11
   ```

   (Output is printed on a single complete line with no padding.) 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 A1Q5: 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`.

   Zip up all of the files that make up your test suite into the file `a2q1.zip`, and submit to Marmoset.

2. In the repository 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.

   In this problem, you will write a program called `change` that makes change for any country's monetary system (real or fictional). This program accepts, as command-line parameters, the coin denominations that make up the monetary system, and the total value. It then prints a report of the combination of coins needed to make up the total, from highest to lowest denomination.

   For example if a particular country has coins with values 1, 10, and 25, and you have 68 units of money, then the command-line would read as follows:

   ```
   ./change 1 10 25 68
   ```

   The initial three values are the coin denominations, in any order. The last value is the total. For this input, the output should be:

   ```
   2 x 25
   1 x 10
   8 x 1
   ```

   Notes:

   - Most coin systems have the property that you can make change by starting at the highest coin value, taking as many of those as possible, and then moving on to the next coin value, and so on. Although not all combinations of coin denominations have this property, you may assume that the input for `change` will always have this property.
   - The Canadian government has recently abolished the penny. Consequently, once the remaining pennies work their way out of circulation, it will be impossible to construct coin totals not divisible by 5. Similarly, in whatever system of denominations you are given, it may not be possible to construct the given total. If that happens, output `Impossible` (and nothing else) to standard output.

- The program needs at least 2 command-line parameters: a minimum of one denomination and one total. If the user doesn't provide at least the minimum number of command-line arguments, output the following line to standard output and exit.

  ```
  Usage: change [denominations] [amount]
  ```

- Valid command-line parameters are positive integers. When testing, you may assume that only valid input is passed on the command-line (i.e. no alphabetic or otherwise invalid characters are passed as arguments).
- Denominations may be listed in any order.
- You may assume that the number of denominations is at most 10. Do not allocate heap memory.
- You may assume that no denomination will be listed twice.
- If a given coin is used 0 times for the given total, do not print it out; your output should contain only those denominations that were actually used, in decreasing order of size.

(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++. Call your program `change.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.

(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. 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,

then `readIntArray` 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.

A test harness is available in the starter file `a2q4.cc`, which you will find in your `cs246/1165/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).

(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 `suiteq4.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q4.zip`.

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

5. The five-letter word game is a logic game similar to hangman and mastermind. A secret five-letter word with no repeating letters is chosen. A player guesses five-letter words (also with no repeating letters). For each word guessed, the player is informed how many letters in the word that they guessed are in the secret word. When the player guesses the correct word, the game is over and they win.

   An implementation of this game should behave according to the following requirements:

   - The program will read a single word (the "secret") from a file specified on the command-line, which is the word that the player should attempt to guess. The syntax for invoking your program should be:

     ```
     wordguess [filename]
     ```

- If invoked incorrectly (i.e. with an incorrect number of arguments), the program should write a message to standard error stating "usage: wordguess [filename]" and exit.
- If a file is specified on the command-line which cannot be opened for any reason, the program should write a message to standard error stating "file cannot be opened", where "file" is replaced with the filename specified on the command-line, and then exits.
- If the secret word has repeating letters, or is not exactly five characters in length, the program should write a message to standard error stating "the secret word is invalid" and exit.
- The game will read a word from standard input (a "guess"), and provide feedback as described below. Players can continue to enter guesses, one word at a time, until the secret word is guessed, or an EOF (Ctrl-D) is entered.
- A valid word consists of five letters, with no letters repeating in a word (e.g. "poach" is a valid word; "speed" is not). Words do not include digits, or symbols, and do not contain spaces (i.e. words containing non-letter symbols are not valid).
- Guesses should consist of lower-case (e.g. "chase" is a valid guess but "chASe" is not), but you do not need to validate or enforce this in your program.
- All string comparisons should be case sensitive (i.e. "SPACE", "space" and "Space" are not the same word). Guesses that differ from the secret only by case of the letter entered are not considered a match (e.g. "claim" doesn't match "clAim").
- If the guess is not equal to five characters in length, or contains duplicate letters, the program should write a message to standard out stating "invalid guess", and allow the player to guess again.
- If a guess does not match the secret word exactly, the game will return a message stating that "x letters match" where x denotes the number of characters in that guess that are also in the secret word. Note that position does not matter; guessing "poach" against the secret word "crash" would match 3 characters, the 'a', 'c' and 'h', even through they had different positions in the two words. The player is allowed to guess again.
- If a word entered matches the secret word, the program will return a message "you guessed correctly!" and exit.

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

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