Assignment #2

Due Date 1: Friday, 31 May, 2019, 5:00 pm
Due Date 2: Friday, 7 June, 2019, 5:00 pm

- **Questions 1, 2a, 3a and 4a 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`
- **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: 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 Assignment 1: each test should provide its input in the file `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: Zip up all of the files that make up your test suite into the file `a2q1.zip`, and submit to Marmoset.

Part 2

In the assignment 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.

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 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 sample executable has been provided. Use it by giving it valid input to confirm that you have understood the expected behaviour from the `change` program.

1. **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`.
2. **Due on Due Date 2:** Write the program in C++. Call your program `change.cc`.

Part 3

In this problem, you will write a program to keep track of student performance in an arbitrary course. The first command-line argument to the program is a non-optional non-empty string representing the course name. The second argument, which is optional, is a number between 0 and 10 inclusive that represents the number of assignments for the course. If the optional argument is not provided, the program defaults to 5 assignments. We use n to represent the number of assignments (whether specified as an argument or the default value of 5). For example, assuming the executable is called `grade`:

- `./grade CS246` executes the program with the course name CS246 and uses 5 for n, the number of assignments.
- `./grade CS241 10` executes the program with the course name CS241 and 10 assignments.

There is always 1 midterm and 1 final exam in the course.
Input to the program comes from standard input and begins with n lines with each line containing the maximum marks per assignment, a space and then the weight of the assignment. This is followed by a line containing the maximum marks for the midterm, a space and then the weight of the midterm. The next line gives the maximum mark for the final exam and the weight of the final exam. You can assume that this part of the input is correct i.e.

- The second command line argument (if supplied) is an integer value between 0 and 10 (inclusive).
- Standard input always begins with n lines indicating assignment maximums and weights.
- Assignment, midterm and final maximums are valid positive integers.
- The total weights of all assignments and exams equals 100%.

After the course information comes a number of student records, each on their own line. You can assume that the input will never contain more than 20 `valid` records. A valid student record contains their userid (a string that does not contain any whitespace) followed by n assignment entries followed by the midterm mark and finally the final exam mark. Each of these are separated by a single space. Each assignment entry must be an integer or the string "ACC" (uppercase) that represents an ACCommodation for the assignment was granted (see below for rules). In addition to these rules, a record is considered invalid if one of the following conditions are true:

- The provided mark for an assignment (or exam) is higher than the maximum mark specified for that assignment (or exam) or negative.
- The number of assignment entries and exam marks provided do not match the total number of assignments and exams

Blank (empty) lines are ignored and not considered invalid records. Input to the program ends when the end-of-file is encountered.
Your program should produce the following output:

- A line containing the name of the course.
- A line for each student record that was invalid. This line contains the userid, a space, and then the string "invalid". Invalid records are listed in the order they were entered.
- A line printing the string "Valid records:" followed by a space and then the number of valid student records.
- A line for each student record which was valid in the order in which they were entered. Each line contains the student's userid followed by a space and then their final mark in the course.
- A line displaying the class average.

For example, assuming the program is executed as: `./grade CS246`, and standard input contains the following data :

```
100 7
100 7
100 7
100 7
150 12
100 20
100 40
nanaeem 34 55 92 80 110 79 88
xy12li 52 69 83 92 119 88 92
abxyz 72 98 110 83 120 56 78
xyz12 89 45 98 100 140 0 92
fyh34j 89 34 12 93 98 39
```

the program should produce the following output:

```
CS246
abxyz invalid
fyh34j invalid
Valid records: 3
nanaeem 74
xy12li 80
xyz12 69
Average: 74
```

Note: the record for abxyz was deemed invalid because the student's assignment 3 marks were recorded as 110 while the maximum mark for this assignment is supposed to be 100. Similarly, the record for fyh34j was deemed invalid because there are only 6 marks provided but there should have been 7 (5 assignments + 2 exams).

Important note on calculating the grade: For this question perform all calculations using variables of type `int`. Using an `int` type will cause rounding to occur. To ensure consistency (and similar rounding errors), you must use the following formula:

```
int mark_assignment = ...
int max_assignment = ...
int weight_assignment = ...
int weighted_assignment = (mark_assignment * weight_assignment ) / max_assignment
```

i.e. always multiply the marks obtained for any given assignment/exam with the weight first and then divide by the maximum marks for that component. Once you have computed the weighted grade for each assignment and the exams, simply add them to get the student's course grade.
Similarly, the average grade (also an `int`) is computed by adding grades of students with valid records and then dividing by the number of valid records.

Assignment accommodations: If an assignment entry is the string "ACC", the weight of that assignment is distributed to **subsequent** assignments. Mathematically, if n represents the total number of assignments and i has been accommodated, then each assignment $i+1$ to n has its weight increased by $weight_i/(n-i)$. In addition, assignment n's weight is further increased by $weight_i\%(n-i)$.

For example, suppose a course has 4 assignments (n = 4) with a weight of 9 for each assignment. Suppose A2 is excused, then the $weight_{A2}$ is distributed between A3 and A4 (not A1). $weight_{A3}$ is increased by $weight_{A2}/(4-2)$ (i.e. 9 / 2 = 4) and $weight_{A4}$ (which is assignment n) is increased by $weight_{A2}/(4-2)+weight_{A2}\%(4-2)$ (i.e., 9/2 + 9%2 = 4 + 1 = 5).

Multiple assignments might be accommodated for a student in which case the first accommodation is used to adjust weights of subsequent assignments and then the next accommodation is used to further adjust the weights of subsequent

assignments based on the adjusted grades. Going with the previous example, where A2 was accommodated, suppose A3 is also accommodated. Then, its adjusted weight of 13 is shifted to subsequent assignments (in this case this is A4). A4 therefore becomes worth 14 (current adjusted weight) + 13 = 27. If the last assignment is accommodated, its (possibly adjusted) weight is moved to the final exam.
Note: Starter code is given to you in `a2q3.cc` in your `a2` directory.

1. **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`.
2. **Due on Due Date 2:** Write the program in C++. Call your solution `a2q3.cc`.

Part 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, `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 reference 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 reference 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 `a2q4.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.

1. **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`.

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