Assignment #2: C++ Basics

Due Date 1: Friday, October 1, 2021, 5:00 pm EST
Due Date 2: Friday, October 8, 2021, 5:00 pm EST
Online Quiz: Monday, October 18, 2021, 5:00 pm EST

Topics that must have been completed before starting Due Date 1:

1. Software Testing
2. `produceOutputs` and `runSuite` from A1

Topics that must have been completed before starting Due Date 2:

1. C++: Introduction to C++
2. Preprocessing and Compilation

Learning objectives:

- C++ I/O (standard, file streams) and output formatting
- C++ `struct`
- C++ `string`s and `stringstream`s
- C++ references, pointers, and dynamic memory allocation

- **Questions 1, 2a, 3a 4a are due on Due Date 1; questions 2b, 3b, and 4b are due on Due Date 2. You must submit the online quiz on Learn by the Quiz date.**
- 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. For each question you will be given a compiled executable program that is a program representing a solution to each question. You should use these provided executables to help you write your test cases, as they can show you the resultant output for given inputs. 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. The size of each input (`.in`) file is also restricted to 300 bytes. This is to encourage you not to combine all of your testing eggs in one basket. There is also a limit for each output file (`.out`), but none of your tests should be able to create such a large output file that you would normally encounter it.
- 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/F21/codingguidelines.shtml`

- We have provided some code and sample executables under the appropriate `a2` subdirectories. **These executables have been compiled in the CS student environment and will not run anywhere else.**
- **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.

Note: We suggest creating the directory `~/cs246/f21/a2` and creating all of your assignment solutions in this directory.

Coding Assessment

Questions 2 to 4 are part of the coding assessment, and may be publicly discussed on Piazza so long as solutions are neither discussed nor revealed.

Question 1

(25% of DD1; 0% of DD2) 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 paid off. If it's not paid off in full, then 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. If any of the numbers contain more than two fixed decimal places, they are rounded to the nearest two decimal places with halfway cases rounded away from zero. For example, 3.159 is rounded to 3.16. 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 A1Q5: 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`.

Notes:

1. Interest rate cannot be negative and should be in the range $\{0,100\}\%$.
2. Previous balance can be negative, 0 or positive.
3. Payment can be 0 or positive but not negative.
4. Try to avoid really large positive or negative values i.e. avoid values above INT_MAX and below INT_MIN.

a) **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.

b) **Due on Due Date 2:** Nothing.

Question 2

(25% of DD1; 30% of DD2)

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 $n = 10$ 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), so you do not need to test for that. 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 ($X = 7$):

```
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/codeForStudents/q2` 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`.

Question 3

(25% of DD1; 30% of DD2)

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 and weights 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 truncating 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 that the midterm and the final exam cannot be accommodated.
Note: '%' is the C++ modulo arithmetic operator and it produces the remainder of an integer division.
Note: Starter code is given to you in `a2q3.cc` in your `a2/codeForStudents/q3` directory.

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++. Call your solution `a2q3.cc`.

Question 4

(25% of DD1; 40% of DD2) 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 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.

A test harness is available in the starter file `a2q4.cc`, which you will find in your `cs246/1219/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.**

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`.

Submission

The following files are due at Due Date 1: `a2q1.zip`, `a2q2.zip`, `a2q3.zip`, `a2q4.zip`.
The following files are due at Due Date 2: `a2q2.cc`, `a2q3.cc`, `a2q4.cc`.