HW: Debugging, Input Validation, and Integer Overflow

This is an individual assignment. Do not share your code with other students. Do not show your code to other students. Do not look at the code of other students. Do not ask other students how they solved a problem. HWs are **individual** practice assignments (LWs, on the other hand, are collaborative practice assignments). If you have questions about this assignment, talk to a TA or an instructor.

If you know about a student who is sharing their code with other students (or in any other way is violating the Aggie Code of Honor), you should report them to the instructor or to the AHSO (make an honor code violation report).

tl;dr (too long; didn't read)

Debug seven "simple" functions. Buggy code and tests are on Mimir.

Requirements

Debug the following seven functions[1]:

1. `Largest` which takes three integer inputs: `a`, `b` and `c`. The function should return the largest of these three values.
    a. Example: `Largest(1, 2, 3)` should return `3`.
    b. `Largest` should not throw an exception.
    c. The function prototype is `int Largest(int a, int b, int c)`.
2. `SumIsEven` which takes two integer inputs and returns `true` if and only if the sum of the two values is an even number. If the sum of the input values is an odd number, then the function must return `false`.
    a. Example: `SumIsEven(3, 5)` should return `true`.
    b. `SumIsEven` should not throw an exception.
    c. The function prototype is `bool SumIsEven(int a, int b)`.
3. `BoxesNeeded` which takes one input. The input value represents a number of apples that need to be packed into boxes. Each box can store a maximum of **20** apples. The function should return the minimum number of boxes needed to store the given number of apples.
    a. Examples:
        i. `BoxesNeeded(13)` should return `1`.

[1] Problems sourced and adapted from Ettles, Andrew & Luxton-Reilly, Andrew & Denny, Paul. (2018). Common logic errors made by novice programmers. 83-89. 10.1145/3160489.3160493.

    ii. `BoxesNeeded(-13)` should return `0`.
  b. `BoxesNeeded` should not throw an exception.
  c. The function prototype is `int BoxesNeeded(int apples)`.
4. `SmarterSection` which takes four inputs and determines which section performed better on a particular exam question. The first input is the number of students in section A that answer the question correctly, and the second input is the total number of students in section A. Similarly, the third input is the number of students in section B that answer the question correctly, and the fourth input is the total number of students in section B. The performance of each section can be measured as the percentage of students who answer the question correctly. The function should return `true` if section A performs strictly better than section B and `false` otherwise.
  a. Example: `SmarterSection(40, 50, 75, 100)` should return `true`.
  b. `SmarterSection` should throw `std::invalid_argument` if the arguments are invalid.
  c. The function prototype is `bool SmarterSection(int A_correct, int A_total, int B_correct, int B_total)`.
5. `GoodDinner` which takes two inputs: the number of pizzas you have eaten, and a Boolean which is `true` if it is the weekend and `false` otherwise. The function should return `true` if you had a good dinner, and `false` if you didn't. A good dinner is a dinner where you eat between 10 and 20 pizzas (inclusive) unless it is a weekend, in which case there is no upper bound on the number of pizzas.
  a. Example: `GoodDinner(13, false)` should return `true`.
  b. `GoodDinner` should not throw an exception.
  c. The function prototype is `bool GoodDinner(int pizzas, bool is_weekend)`.
6. `SumBetween` which takes two integer inputs, `low` and `high`. This function should calculate the sum of all integers between low and high, inclusive.
  a. Example: `SumBetween(1,10)` should return `55`.
  b. `SumBetween` should throw `std::invalid_argument` if `low` and `high` are out of order.
  c. `SumBetween` should throw `std::overflow_error` if the sum exceeds the maximum/minimum value of `int`.
    i. `INT32_MAX` is the maximum value of a 32-bit integer.
    ii. `INT32_MIN` is the minimum value of a 32-bit integer.
    iii. See Integer Overflow for more explanation.
  d. The function prototype is `int SumBetween(int low, int high)`.
  e. Try to only use `int`. Don't use a bigger type (it can work, but is not in the spirit of the problem).

  HINT 1: Suppose `a` and `b` are two positive integers such that `(a + b)` results in an overflow. To form conditions for `overflow_error`, consider using subtraction

expressions (`INT32_MAX - a` or `INT32_MAX - b`) instead of addition. Likewise, think about forming conditions with `INT32_MIN`.

HINT 2: Consider how you could modify `low` and `high` so that you can avoid overflow during calculating the sum. For example, if `low = INT32_MIN` (`-2147483648`) and `high = INT32_MAX` (`2147483647`), there is no need to sum at all (the "sum" is simply `-2147483648`).

7. `Product` which takes two integer inputs: `a` and `b`. The function should return the product of these two values.
    a. Example: `Product(2, 2)` should return 4.
    b. `Product` should throw `std::overflow_error` if the product exceeds the maximum/minimum value of int.
        i. `INT32_MAX` is the maximum value of a 32-bit integer.
        ii. `INT32_MIN` is the minimum value of a 32-bit integer.
        iii. See Integer Overflow for more explanation.
    c. The function prototype is `int Product(int a, int b)`.
    d. Try to only use `int`. Don't use a bigger type (it can work, but is not in the spirit of the problem).

    HINT: Consider forming conditions for `std::overflow_error` by using `a`, `b`, and `INT32_MAX` using division expressions instead of multiplication. Likewise, use `a`, `b`, and INT32_MIN in division expressions (for negative product values).

General Instructions

Each of the seven functions already has a buggy implementation provided on Mimir.

1. Get Starter Code

A convenient way to compile multiple source files (in this case the two files: `functions.cpp` and `debugging.cpp`) is to put all the source and header files in a directory (for example, `hw_debugging`) and run `g++` on all source files in this directory:

- Download the starter code from Mimir.
- Make a directory named `hw_debugging`.
- Put the starter code in `hw_debugging`.

2. Remove Syntax Errors

In a terminal (assuming you are in the parent directory of `hw_debugging`):

```
$ cd hw_dubugging
$ ls
debugging.cpp  functions.cpp  functions.h
```

```
$ g++ -std=c++17 -Wall -Wextra -Weffc++ -pedantic -g *.cpp
```

The * symbol is a wildcard that matches any valid character in an identifier. So, `*.cpp` means all files in the current directory whose name ends in `.cpp` (i.e. all C++ source files).

- Correct the syntax error(s), if any. Don't worry about warnings for now since warnings don't prevent successful compilation of the code. Most warnings should be resolved as you debug each function
    - Note that a syntax error can get the compiler off track. So, it might end up identifying things as errors that are not really errors. So sometimes when you **fix the first syntax** error and recompile, many if not all of the subsequent errors are resolved!
    - Fix the **first** compiler error and then re-compile.
        i. Repeat until there are no syntax errors.

3. Debug Functions

Do not completely rewrite the functions. Try to make the fewest and smallest changes possible to fix the errors. Only if it seems that the implementation is doomed to fail should you consider trashing and rewriting the code.

- The problems generally get a little bit harder as you go from problem 1 to problem 7. You don't have to go in order, but you probably should.
- Go through each function and debug one function at a time using the Requirements listed above.
    - You will have to think carefully about what the requirements are and come up with test cases that you can use to debug.
    - We have provided a main function (in `debugging.cpp`) that you can use to input test values for each function. You can use the provided main. You are also free to edit and/or replace it with your own custom test code to call the functions. None of the test cases rely on the provided main function.
    - Do not try to read from standard input in the functions. Nothing will come. Your program will starve and timeout. That will be sad. The input comes from the function parameters. You can specify the input by writing test cases in the main function or using the existing main function and providing values by hand when you run it locally.
- When you feel you have a function working, submit your code to Mimir and see if you pass the test cases for that function.
    - Click the tests for more details. Every test will give you useful information.
- If you don't understand an error or get stuck, try searching for the error message first, then contact your TA or instructor. Do not spend more than about 20 minutes stuck before asking for help.

- Stuck means making no progress. You may have an idea of what to do and need to read some documentation to figure out how to do it, which might take several hours. That's OK. You're not stuck… yet.
    i. For example, you'll need to know how to detect arithmetic overflows and throw exceptions. Learning this will take some time. That time spent learning is not time spent being stuck. However, if, once you are attempting to implement a fix, you spend more than about 20 minutes attempting to debug the code you wrote to fix the original error, you now are stuck. Get help.

Have Fun!

Appendix: Integer Overflow

Integers in C++ come in various sizes:
8 bits, e.g. `char (int8_t)`
16 bits, e.g. `short (int16_t)`
32 bits, e.g. `int (int32_t)`
64 bits, e.g. `long (int64_t)`

Each of these types of integers can store only a certain range of numbers, based on their size and signed-ness (totally a word… I used it, you knew what I meant, it's a word.).
8-bit integers store integers in the range [-128, 127] or [0, 255] (unsigned)
16-bit integers store integers in the range [-32768, 32767] or [0, 65535] (unsigned)
32-bit integers store integers in the range [-2147483648, 2147483,647] or [0, 4294967295] (unsigned).
64-bit integers store integers in the range [-9223372036854775808, 9223372036854775807] or [0, 18446744073709551615] (unsigned).

For convenience, C++ has names for these constant limits; see them all at en.cppreference.com. For example:

```
#define INT8_MIN -128
#define INT8_MAX 127
```

Signed integers have one bit reserved to encode the sign of the number, while the rest of the bits are used to encode the magnitude. Unsigned integers use all the bits for the magnitude. This is why unsigned numbers can represent "bigger" numbers: they sacrifice negative numbers in exchange for bigger positive numbers.

This is an example of a 8-bit signed integer (i.e. `char`, this one is 65, or 'A',):

```
0 1 0 0 0 0 0 1
^ ^-----------^
|  magnitude bits
sign bit
```

If we add 1 to 65 (‘A’), we get 66 (‘B’):

```
  0 1 0 0 0 0 0 1
+               1
-----------------
  0 1 0 0 0 0 1 0
```

If we keep adding 1, we eventually reach 126 (‘~’) and then 127 (DEL, not printable).

```
  0 1 1 1 1 1 1 0  // 126
+               1
-----------------
  0 1 1 1 1 1 1 1  // 127
```

If we add 1 to 127, we get 128, right?

```
  0 1 1 1 1 1 1 1  // 127
+               1
-----------------
  1 0 0 0 0 0 0 0  // 128?
```

Actually… no. Since our integer is signed, we get -128.

```
1 0 0 0 0 0 0 0  // -128
^ ^-----------^
|  magnitude
sign
```

That looks like -0, right? It's not. Signed integers are represented using something called 2's complement. Don't worry about how it works right now (it's actually simple… but also out of scope for this class).

The important thing to note is that the magnitude of the number ***overflowed*** into the sign bit. A similar thing happens to unsigned numbers that get too big, except the overflow just falls off the end and is lost:

```
  1 1 1 1 1 1 1 1  // unsigned 255
+                1
------------------
1|0 0 0 0 0 0 0 0  // unsigned 0
^
overflow is lost
```

This problem, where a mathematical operation on two integers results in a number which does not fit into the data type (e.g. two 8-bit numbers added together can sometimes yield a 9-bit number) is called ***Integer Overflow***. It can be a serious problem in C++ programs.

Whenever we do math with integers where the result could be larger in magnitude than the data type has room for, we must guard against integer overflow. The result of an operation which overflows is undefined. Usually it's whatever actually did fit (the part that overflowed is lost), but it may not be. When the type is signed, the overflow sometimes flips the sign bit (but not always). Because of this, detecting an overflow cannot be done by simply comparing the result of the operation to the limit:

```
// this will NOT work!
char tilde = 126;
char three = 51;
if (tilde + three > INT8_MAX) {
    // handle overflow
}
```

126 + 51 = -79 for signed 8-bit integers (e.g. `char`). And -79 is not > 127.

So, how can you do it?

That is left as an exercise for the student.