HW: Debugging

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 Peer Teacher, a TA, or an instructor.

If you know about another 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](https://cm.maxient.com/reportingform.php?TexasAMUniv&layout_id=11)).

# 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]](#footnote-0):

1. Product() which takes two integer inputs: a and b. The function should return the product of these two values.
   1. Example: Product(2, 2) should return 4.
   2. Product should throw std::overflow\_error if the product exceeds the maximum/minimum value of int.
      1. INT32\_MAX is the maximum value of a 32-bit integer.
      2. INT32\_MIN is the minimum value of a 32-bit integer.
      3. See [Integer Overflow](#_fhrhdx8rxkll) for more explanation.
   3. The function prototype is int Product(int a, int b).
   4. Try to only use int. Don't use a bigger type (it can work, but is not in the spirit of the problem).
2. Largest() which takes three integer inputs: a, b and c. The function should return the largest of these three values.
   1. Example: Largest(1, 2, 3) should return 3.
   2. Largest should not throw an exception.
   3. The function prototype is int Largest(int a, int b, int c).
3. 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.
   1. Example: SumIsEven(3, 5) should return true.
   2. SumIsEven should not throw an exception.
   3. The function prototype is bool SumIsEven(int a, int b).
4. SumBetween() which takes two integer inputs, low and high. This function should calculate the sum of all integers between low and high, inclusive.
   1. Example: SumBetween(1,10) should return 55.
   2. SumBetween should throw std::invalid\_argument if low and high are out of order.
   3. SumBetween should throw std::overflow\_error if the sum exceeds the maximum/minimum value of int.
   4. The function prototype is int SumBetween(int low, int high).
   5. Try to only use int. Don't use a bigger type (it can work, but is not in the spirit of the problem).
5. 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.
   1. Examples:
      1. BoxesNeeded(13) should return 1.
      2. BoxesNeeded(-13) should return 0.
   2. BoxesNeeded should not throw an exception.
   3. The function prototype is int BoxesNeeded(int apples).
6. SmarterSection() that takes four inputs. 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 a question correctly in class. The SmarterSection() function should return true if section A performs better than section B and false otherwise.
   1. Example: SmarterSection(40, 50, 75, 100) should return true.
   2. SmarterSection should throw std::invalid\_argument if the arguments are invalid.
   3. The function prototype is bool SmarterSection(int A\_correct, int A\_total, int B\_correct, int B\_total).
7. 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.
   1. Example: GoodDinner(13, false) should return true.
   2. GoodDinner should not throw an exception.
   3. The function prototype is bool GoodDinner(int pizzas, bool is\_weekend).

# Starter Code

Each of the seven programs already has a buggy implementation and more description provided on Mimir. Each program is a separate problem/question on the Mimir homework assignment. Some of them have syntax errors that should be fixed before attempting to debug the logic of the function.

# General Instructions

1. Start by correcting the syntax error(s), if any.
2. 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.
3. Every problem has a “Description” tab that describes the problem, gives an example, and specifies the function prototype.

[HW] 3. Debugging
1. Product
The description tab is selected.
Debug a function called Product() which takes two integers inputs: a and b.  The function should return the product of these two values.
Example: Product(2, 2) should return 4.
The function prototype is int Product(int a, int b).
START by correcting the syntax error(s), if any.

1. Every problem has a “Grading” tab, which contains tests for checking the correctness and style (sort-of) of the code. You should run these tests before making any modifications to the code.
   1. The details of the tests are mostly hidden. You should construct your own test cases to use to debug each function.

[HW] 3. Debugging
1. Product
The grading tab is selected.
Red bar indicating all tests failing.
White "Collapse Results" button.
Green "Run Tests" button.
Test: Hidden happy path test cases 
0 / 5 pts - Click for details
FAILED
Test: Hidden unhappy path text cases FAILED
0 / 5 pts - Click for details
FAILED
Test: lint FAILED
0 / 0 pts - Click for details
FAILED
Test Product(2, 2) FAILED
0 / 5 pts - Click for details
FAILED

1. Every problem has a “My Test” tab where you can write your own unit tests. You may find this useful as a way to “run” your code.
   1. None of the functions require you to print anything. Nor do the tests care if you do -- all output will be ignored.
   2. If you need to print something to standard output you should remember that cout and endl’s real names are std::cout and std::endl.
      1. Use using std::cout, std::endl; if you are too lazy to type the namespace and scope resolution operator: std::.
   3. Do not try to read from standard input. 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.

[HW] 3. Debugging
1. Product
My test tab is selected.
Main file name: code.cpp
C++17 unit code (GTest and GMock asserts)
Code box containing a full screen button, a settings gear button, code as follows:
line 1: // example test:
line 2: EXPECT_EQ(Product(2, 2), 4);
Green "Run Test" button
Test: Write Your Own Tests
Click for details
Completed

1. Click the tests for more details. Every test will give you useful information.
   1. Hint: The inputs of the hidden test cases can be borrowed by printing them to the console.
2. 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.
   1. Problem 1 (Product) is actually not easy, so maybe start with Problem 2 (Largest)
3. 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.
   1. 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.
   2. 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!**

# 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 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](https://en.cppreference.com/w/cpp/types/integer). 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 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.

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. [↑](#footnote-ref-0)