# Homework 1

**Assigned:** 8/31

**Due:** 9/7, 11:59pm

**Possible Points:** 100

Submit on Canvas a zip or zipped tar file containing your solution code for each problem (just the cpp files, please don't submit the solution/project files), turn in one cpp file for each problem. If a problem has multiple parts, it should still be one file. Each problem specifies what you should title the file containing your solution so that the autograder can find it. Incorrectly named programs will not be graded and thus will get 0 points for the problem.

Since the assignment will be graded using an automated grading system you should have your program's output match the sample output shown. If your program fails to compile using Visual Studio 2015 it will recieve zero points. If your program compiles but fails to run properly (runtime errors or wrong output) points will be deducted based on how correct the program is.

If you have any questions or concerns about the assignment do not hesitate to post on the discussion board on canvas, send us mail on canvas, or see us after class or during office hours.

Students are expected to write their source code from scratch, those who copy code from each other or online (including from Canvas examples) will be reported for cheating. We will also look at your code so definitely don't just hard-code the answer and print it.

## 0. Testing Cases: 10 points

**File:** `testing_cases.cpp`

This program will handle a simple situation of reading the number of cases to be tested and will then iterate through each case, printing the case number as `Case #:` and echoing back the word input by the user. Your case numbers should start counting from 0. The case number printed out is a header so our autograder knows when to start grading a new case. The echoing of the user input is the code being tested by the grader.

For this problem, you should use `std::cin` and the `>>` operator to read strings of characters from the standard input, and store them in variable(s) of type `std::string`. For output, use `std::cout` and the `<<` operator. Remember to `#include` the `<iostream>` and `<string>` header files from the standard library for this to work. For convenience, you may want to save the input in a text file and not type it every time you want to test your program. For this, you will

need to redirect the standard input to a file. See the Day2's notes on Canvas for how to do this.

**Important notes** For the rest of the assignments in this homework (and future ones) we will always begin the input by the number of test cases (which is always a positive integer), but the input format for the test cases vary from problem to problem. Here, for each problem we provide a sample input (or Test Set) for you to test your program, but your program will be tested using our own test set not limited tothose shown here, so try to make your code robust (e.g. pay attention to uncommon inputs and edge cases). Finally, make sure the format of your output follows that of the expected output's exactly.

**Example Input**

```
7
Hello!
Goose Chicken Duck
Hello
Computer
Whoops!
```

**Expected Output**

```
Case 0:
Echo: Hello!
Case 1:
Echo: Goose
Case 2:
Echo: Chicken
Case 3:
Echo: Duck
Case 4:
Echo: Hello
Case 5:
Echo: Computer
Case 6:
Echo: Whoops!
```

---

## 1. Converting temperatures: 20 points

**File:** `celsius_fahrenheit.cpp`

For this assignment you will write a program to compute the equivalence in Celsius for Fahrenheit temperatures, using `int` and `double`. Your program will receive as input a temperature in Fahrenheit and should print out the equivalent

temperature in Celsius, first using `int` then using `double`. The input will always be an integer.

Remember the first line of input is always the number of tests.

**Example Input**

```
4
48
70
105
90
```

**Example Output**

```
Case 0:
48F = 8C
48F = 8.88889C
Case 1:
70F = 21C
70F = 21.1111C
Case 2:
105F = 40C
105F = 40.5556C
Case 3:
90F = 32C
90F = 32.2222C
```

---

## 2. 80's Game Development: 40 points

**File:** `game_development.cpp`

For this problem you'll pretend to be a game developer from the 80's, trying to prevent people from healing your boss to death. Unfortunately due to memory constraints you can only use a `short` to store the boss's health, which starts at 32000. The player will attack or heal the boss once. You will have to make sure the boss cannot be healed to death due to overflow. Note that it can still be attacked to death. If you detect an overflow you should print out:

```
no healing the boss to kill it!
```

Whether you detect an overflow or not, print the health of the boss after the attack/heal attempt.

```
boss health is XXXXXX
```

The boss's health is capped so that it should always be non-negative, and less than or equal to the maximum number represented by the type `short`. If a

healing attempt has the potential to bring its health pass the maximum value, the boss's health stays at the maximum value. Similarly, an attack can only bring the boss's health to 0 at the minimum.

If you detect that the boss is dead, because its health is 0, print

```
the boss is dead!
```

The input to your program will be an integer in the range of the `short` type. Attacks will be negative, reducing the boss's health and heals will be positive, increasing the boss's health. Note that the boss's health will reset to its default (32000) for each new test case.

**To get full credit for this problem you must prevent the overflow from happening at all, not allow it to happen and then try to revert it. Also you will be penalized if you wastefully use a larger type than `short` to store the boss's health, as memory is scarce in the 80's.**

**Example Input**

```
5
200
-1000
800
-32000
-32001
```

**Example Output**

```
Case 0:
boss health is 32200
Case 1:
boss health is 31000
Case 2:
no healing the boss to kill it!
boss health is 32767
Case 3:
boss health is 0
the boss is dead!
Case 4:
boss health is 0
the boss is dead!
```

---

## 3. Preventing Floating Point Overflow: 30 points

**File:** `preventing_overflow.cpp`

Write a program to check if casting a double to a float would cause overflow to infinity, your program should check that the number being assigned is within the range of values that a float can store (hint `std::numeric_limits<float>`). Your program should read six double-precision inputs each of which you should test if they're safe to assign to a single-precision float.

Your program should output the following if the assignment would overflow (`XXX` is the number we're trying to assign):

```
XXX will overflow a float
```

In the case that it wouldn't overflow you should print that it won't overflow and the result of the assignment in a float.

```
XXX won't overflow a float, float = XXX
```

The input for each test case will be a real number. Do not worry if the input number is written in scientific format (e.g., `1e20`), using `std::cin` with `>>` as usual takes care of all formats for you.

**Example Input**

```
6
2e5
100.0
-124.05012
1e39
-1e40
-1e20
```

**Example Output**

```
Case 0:
200000 won't overflow a float, float = 200000
Case 1:
100 won't overflow a float, float = 100
Case 2:
-124.05 won't overflow a float, float = -124.05
Case 3:
1e+39 will overflow a float
Case 4:
-1e+40 will overflow a float
Case 5:
-1e+20 won't overflow a float, float = -1e+20
```