The program is broken up into two major parts: The jug class and the calculation. The Jug has five public functions and two private variables.

Jug::setJugMax()

Allows the physical dimension of the jug to be set.

Jug::isJugEven()

Returns a bool of whether the jug is an even amount or odd amount (Ex: a jug that can hold 6 returns true, a jug that can hold a maximum of 5 returns false)

Jug::setJugCurrent(int val)

Sets the amount of water in the jug. If the amount of water in the Jug is greater than the amount of water the jug can hold, it is automatically set to the maximum the jug can hold. (EX: If there’s 11 liters when the jug can only hold 6, then the jug will be set to 6.)

Jug::getJugCurrent()

Returns amount of water in a Jug.

Jug::getJugMax()

Returns the maximum a jug can hold.

Jug::jugCurrent

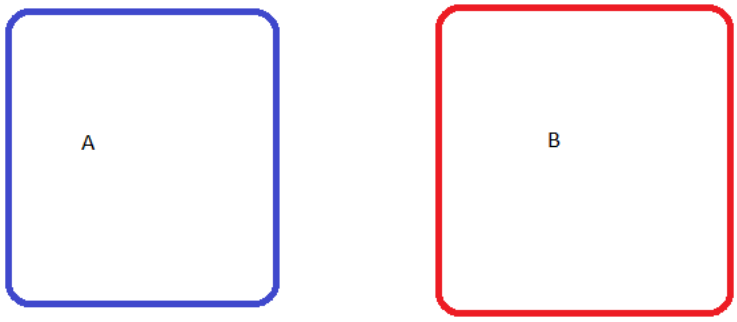
Amount of water currently in a jug

Jug::jugMax

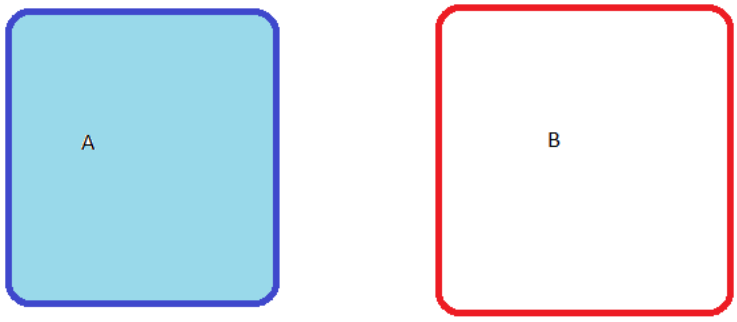
Maximum amount of water a jug can hold.

The main function returns the steps necessary to reach the goal, given the two jugs dimensions. Initially, the user is prompted to enter in the maximum the first and second jug can hold. Next, the user is prompted to enter the goal. The smaller jug is always set to jug A, and the larger jug is set to jug B.

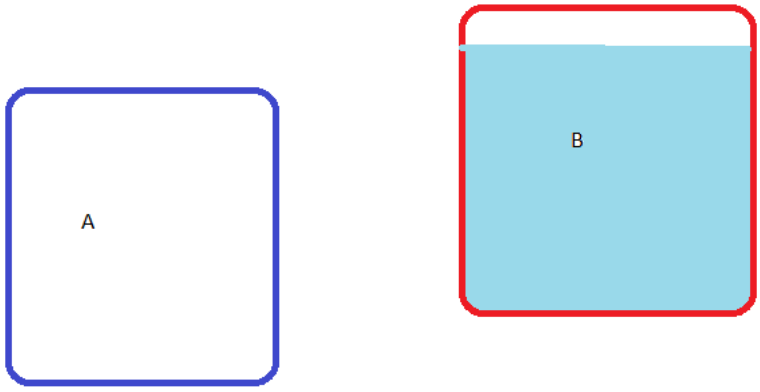
While attempting to find whether the goal is an actual plausibility given the two jugs dimensions, the first test is to see whether the goal is larger than both jugs combined. Obviously, if both jugs, combined, could only hold 30, then it would be impossible to hold 50. Furthermore, another check on whether both jugs are even and the goal is even. If, the goal was 3, but the jugs are 2 and 4, it would be impossible to get 3. For the actual calculations of the jugs, this program uses a simple technique of pouring the smaller jug into the larger jug.



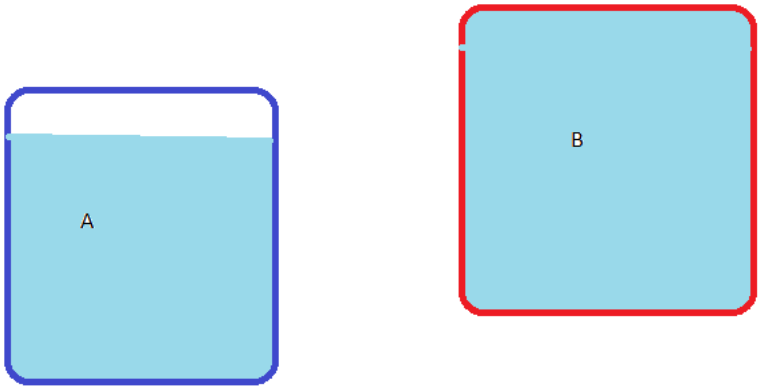
The first loop is a while loop that continues until the addition of water in both jugs is equal to the goal set by the user. Within this while loop, we begin filling up the jugs. First, we check and see if any water is in jug A. If there is, we pour it into B. If not, we fill it, then pour into B. We then print these instructions to the terminal.



After changing the value of jug A, we check to see if we've made our goal with the combination of Jug A and Jug B. If not, we continue. Otherwise, our program ends. Assuming we haven't, we check to see if jug B is filled. If Jug B is filled, we empty it. Otherwise, we continue.



Next, we have a while loop to pour water from jug A into jug B. Until either jug A is empty, or jug B is full, we continue pouring water. After pouring the water, we print these instructions to the terminal. This is the last instruction of the loop that contribute towards finding the goal, and the while loop checks to see if the sum of jug A and B has reached the goal.



If our goal hasn't been reached, fill up A if it's empty, check if we've reached our goal, pour A into B, then empty out jug B. Continuing the while loop cycle.

Our program has a limit of 500 attempts before requesting whether the user wants to continue. For every "y" the user types, that's an extra 500 attempts the program will continue. The user can type "y" multiple times on one line for 500 * y attempts. If, for example, the user has a 1 jug and a 1000 jug, with a goal of 799, this would allow the user to continue, as that is possible. If the user types "n," the program ends.

"g++ main.cpp -std=c++14 -O3"

The *-std=c++14* eliminate compiler warnings, while *-O3* heavily optimize the program, allowing it run much faster.