NAME: _____

Learning Objectives

- Learn how to implement classes (also called user defined data types).
- Understand the concept of encapsulation (hiding internal details to make usage of a class more obvious).
- Understand how to implement and use accessor and mutator functions.
- Understand how to implement constructors.
- Understand how to overload member functions.
- Learn how to break a large program into separate files (also called compilation units).
- Learn how to solve programming problems with classes.

Instructions

Work out the answers to these problems manually without the use of a computer. This will help you develop the ability to read and analyze code.

Another reason to practice solving these problems manually is that problems of this type will appear on Quiz 2 and subsequent quizzes and exams. During the quiz/exam, you will not have access to a computer to solve these problems, so you need to develop the ability to read code, analyze it and think critically.

After coming up with answers manually, you can write and run test code to check whether you have solved the problems correctly.

Submit a single document with your answers either by email or through your remote git repository. If submitting through git, your document should be located in a folder named *a2*.

Problems

```
#include <iostream>
#include <cassert>

using namespace std;

/*
   EXERCISE 1: Draw the UML class diagram for
               the BeanJar class.
*/
```

```
/*
Instances of the BeanJar class represent bean jars of varying
capacity. The class provides functions to add beans, remove beans,
and get the number of beans that are currently in the bean jar.  The
functions to add and remove beans return a boolean value that
indicates whether the operation succeeded or failed.  When one of
these functions fails, the number of beans in the jar will not have
been changed. For example, if a bean jar has a capacity of 10 beans
and there are 9 beans currently in the jar, then calling addBeans(2)
will return false and the number of beans in the jar will remain at
9.
*/

class BeanJar
{
public:
    BeanJar(int maxNumberOfBeans, int initialNumberOfBeans);

    // addBeans returns true on success, false on failure.
    bool addBeans(int numberOfBeansToAdd);

    // removeBeans returns true on success, false on failure.
    bool removeBeans(int numberOfBeansToRemove);

    // getNumberOfBeans returns the number of beans
    // in the bean jar.
    int getNumberOfBeans();

private:
    int maxNumberOfBeans;
    int currentNumberOfBeans;
};

BeanJar::BeanJar(int maxNumberOfBeans, int initialNumberOfBeans)
{
    this->maxNumberOfBeans = maxNumberOfBeans;
    this->currentNumberOfBeans = initialNumberOfBeans;
}

bool BeanJar::addBeans(int numberOfBeansToAdd)
{
    // PROBLEM 2: implement the addBeans function.
}
```

```
bool BeanJar::removeBeans(int numberOfBeansToRemove)
{
    if (numberOfBeansToRemove > numberOfBeans)
    {
        return false;
    }
    else
    {
        currentNumberOfBeans -= numberOfBeansToRemove;
        return true;
    }
}

int BeanJar::getNumberOfBeans()
{
    return currentNumberOfBeans;
}

int main(int argc, char * argv[])
{
    BeanJar beanJar(10, 5);
    assert(beanJar.getNumberOfBeans() == 5);

    assert(beanJar.addBeans(3));
    assert(beanJar.getNumberOfBeans() == 8);

    assert(!beanJar.addBeans(4));
    assert(beanJar.getNumberOfBeans() == 8);



    // PROBLEM 3: Add code to test the removeBeans function.
    // Make sure the test code executes all lines of code
    // in the removeBeans function.



    cout << "All tests passed.\n";
}
```

Figure 1

1) Draw the UML class diagram for the BeanJar class given in Figure 1. (10 points)

2) The operation of the BeanJar class is described in a comment that appears in the code in Figure 1; make sure you read this comment carefully to understand the class. Provide an implementation of the addBeans function of the BeanJar class. (10 points)

3) Implement the missing test code from the main function that appears in Figure 1. Make sure the test code *provides good coverage* in the sense that it executes all lines of code in the removeBeans function. (10 points)

4) Provide an alternative implementation of the addBeans function that sets the number of beans in the jar to its maximum when the added beans exceed the available capacity. The function should continue to return false in this case. (10 points)

5) Provide an alternative implementation of the removeBeans function that sets the number of beans in the jar to zero when the beans to remove are greater than the number of beans in the jar. The function should continue to return false in this case. (10 points)